A PROLOG IMPLEMENTATION OF A SUBSET OF MARCUS' PARSER AND ITS RELATION TO THE HANDLING OF EXTRAGRAMMATICAL INPUT

By

MICHAEL SCARLETT DOROTICH

B.Sc., University of Saskatchewan, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1986

©Michael S. Dorotich, 1986

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date August 21, 1986

Abstract

In any system employing a natural language interface, there is the problem that, by means of a formal grammar, the system itself defines the language it will accept. But, when using language, people will not always adhere to the rules of this grammar; therefore, a natural language computer system should not simply treat as incomprehensible any input not conforming to its internal grammar, input we may call *extragrammatical.* The term extragrammatical refers to input that is not necessarily incorrect in an absolute sense but only relative to the formal scope of a system's grammar. Before a truly robust system can be developed, what is needed is a parsing mechanism that enforces grammaticality where possible, and this implies a deterministic approach to natural language parsing. This thesis discusses the importance of flexible natural language interfaces; the notion of extragrammatical language and its connexion to robust parsing; a deterministic parser, **PARSIFAL**, developed by Mitchell Marcus; and a reimplementation, using logic programming, of a subset of Marcus' system. Programming was done with **CProlog** on a VAX 11/750* running 4.2 BSD UNIX.†

*** VAX is a trademark of Digital Equipment Corporation.**

† UNIX is a trademark of AT&T Bell Laboratories.

Table of Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Harvey Abramson, for his suggestions and comments throughout the writing of this thesis, and Richard Rosenberg for his gracious consent to be the second reader. I must express my gratitude to the National Sciences and Engineering Council for financial support. Finally, I would like to extend many thanks to Linda Plumb for her moral support and encouragement throughout my time as a Master's student.

Chapter 1
Introduction

Considerable work has been done on the design of natural language man-machine interfaces. In most informal settings, people use language without adhering to strict grammatical conventions—even if they have an unconscious operative grammar in their minds—but deviations are inherent in spontaneous language use whatever the modality, even in highly constrained formal settings. Now, if computers are to understand human language, they must parse as robustly as humans do. A computer system should not simply treat as incomprehensible any input that does not conform to its internal grammar. Systems in which this is a concern include database and expert systems.

Natural Language Understanding (**NLU**) systems and Natural Language Interfaces (**NLI**'s) are comprised of several components, the most fundamental being the grammar used to describe input. Most systems, however, are not equipped with mechanisms that attempt to handle input rejected by strict grammatical processing. Nevertheless, people customarily communicate with less than perfectly constructed sentences and naturally expect of computers some of the skills in understanding they themselves exhibit. This may become increasingly the case as artificial intelligence techniques, applied to user interfaces, give these interfaces more and more the semblance of intelligence. When a system intelligently answers questions or offers criticism, a user will naturally assume intelligent conversational abilities of the system. Problems arise when the user does not accurately perceive the limits of the system; therefore, a system should present the user with a consistent model of its capabilities. Moreover, within an educational expert system setting, there are certain fundamental pedagogical reasons for having a sophisticated natural language interface.

In the mid 1970's, **NLU** systems designers noted that understanding requires some attempt to interpret, not merely reject, what *seems* to be ill-formed input. Subsequently, work was done on seeing how the then current natural language parsing mechanisms might handle such input. Recently, a new tool for designing **NLI**'s has emerged: logic programming. It would be interesting to investigate how an **NLU** system founded upon logic programming might be made to perform reasonably in the face of ill-formed input.

In any **NLU** system, there is the problem that the grammar itself defines the language the system accepts. Input may deviate from that which is acceptable either because it is wrong or because the grammar itself is wrong or incomplete. So the first requirement of an **NLU** system that handles seemingly ill-formed, or extragrammatical, input is a parser that enforces grammaticality where possible but behaves gracefully where not, accepting sentences that do not fit the grammar and noting the ways in which they are deviant. In order to do this, a parser must recognize immediately that a sentence has deviated from its grammar. A standard top-down parser would not work. To see why not, consider that when such a parser gets stuck it takes this to mean it has made an incorrect decision earlier in the parse of a given sentence whereupon it backs up and tries an alternate parse. Failing all possible parses, the parser simply gives up and cannot tell where the difficulty lies. This is a problem not only for natural language parsers but also for formal computer language parsers. In fact, this is what led the early recursive descent approach to be replaced by top-down *LL*, or bottom-up *LR*, deterministic approaches. What is needed, then, is a deterministic natural language parser and for this one may turn to the work of Mitchell Marcus.

This thesis involves a survey of the literature pertinent to robust natural language interfaces and a reimplementation in **Prolog** of a subset of Marcus' deterministic parser, **PARSIFAL**. It is assumed that **Prolog** is already familiar to the reader.[1] Likewise, the reader is assumed to be familiar with elementary grammar theory.[2]

The importance of sophisticated natural language interfaces in database and computer assisted instruction systems is investigated in chapter two. Of major significance to the design of a truly flexible natural language understanding system is the handling of extragrammatical input, input that does not conform to the system's grammar. This is the subject of chapter three. The first step towards handling unexpected input is a deterministic parser, and so the work of Mitchell Marcus is discussed in chapter four. Chapter five gives a brief look at logic programming, and chapter six presents a **Prolog** version of part of the **PARSIFAL** system. Chapter seven discusses limitations both of this work and that of Marcus' on which it is based. Conclusions are drawn in the last chapter. Source code and sample parses may be found in the appendices.

[1] The standard reference is [CLO81].
[2] This can be found in compiler writing books like [TRE85].

Chapter 2
Natural Language Interfaces

NLU systems usually fall into one of two categories: those for studying natural language phenomena in general, and those tailored as interfaces to a particular task domain within database and expert systems.

Two influential systems developed in the early 1970's attempted to tackle both syntactic and semantic aspects of natural language. **SHRDLU** [WIN73, WIN80], representing the first category, was designed to show that in order to understand language, a program must integrate syntactic processing, semantic processing, and reasoning.[3] **LSNLIS** [WOO72], on the other hand, was designed to help geologists analyze moon rock samples using a database at the National Aeronautical and Space Agency.

Both of these systems are based on procedural representations of language. The philosophy behind **SHRDLU** is that language activates procedures within the hearer. Thus, syntax and meaning can be represented directly as executable computer programs while reasoning corresponds to the actual execution of the programs. **LSNLIS**, one of the first systems to employ augmented transition networks (**ATN**'s) [WOO70], operates by translating English queries into a formal query language. The formalized query is then presented to the database for an answer. Another such system is **PLANES** [WAL78].

Alternatives employed by other researchers are declarative representations of which the most outstanding are logic and semantic networks. Ideas behind semantic networks have been employed in a number of **NLU** systems including **RENDEZVOUS** [COD74] and **LADDER** [HEN78]. Like procedural

[3] In [WIN80], Winograd reviews SHRDLU and discusses further directions he has taken in natural language understanding.

representation based systems, these also translate English queries into a data sublanguage.

At one time there was a controversy among researchers as to whether procedural or declarative representations should be used. This has died, however, but it is interesting to note that with the advent of logic programming both procedural and declarative interpretations may coexist [CLO81, KOW79, ROB83].

There are a few **NLU** systems—call them *metasystems*—that may be used to aid building other systems which fall into either of the two categories mentioned above. **ProGrammar** [SAL85] and **SAUMER** [POP84], for example, may be used to build and test grammars that may in turn be used for linguistic analysis. **LIFER** [HEN77], which includes a grammar editor, is a utility for building natural language front ends targeted to any domain.

Discussion in this and subsequent chapters will refer mainly to application systems and linguistic phenomena that arise in them.[4] This chapter discusses the need for flexible natural language interfaces in database and educational expert systems.

2.1 Flexible Natural Language Interfaces in Database Systems

The motivation one finds in the literature for natural language access to databases is similar from one system to another. Natural language is convenient and familiar to all and to a casual user is an easier means of making a query than some special formal language or menu. If the user wishes to display elements satisfying several predicates which require logical combinations of multiple files, a menu is not even sufficient: a special data language is required. But learning

[4] For a good overview of all types of natural language understanding systems see [BARR82].

such a language or, at least, using someone as an intermediary presents an obstacle to a nontechnical person. Not only might learning a new language be difficult but, because a person is used to thinking in his native language, learning to translate into the new language might be difficult too. A nontechnical person cannot be expected to be knowledgeable about computers, programming, logic, or relations and yet must be able to obtain information with a minimum of training.

In order for an **NLI** to be of value it must have a large vocabulary of the subject matter, accept a wide range of grammatical constructs, feed back understanding of requests, tolerate spelling and simple grammatical errors, and allow addition of new words and grammatical constructs to the knowledge base. Furthermore, because a user will often ask several questions about the same object, it is convenient that he be allowed to enter elliptical constructs, incomplete input fragments, or pronominal references and that the system interpret them in the context of previous input. While these are commonly set as objectives by **NLI** designers, not all of them are met in all systems and there is certainly room for further research into natural language understanding.

The **RENDEZVOUS** [COD74] system was designed with the intent of having a user engage a relational database system in dialogue to attain mutual agreement about the user's needs. Codd, who designed the system, states that earlier systems failed because they assumed that if a user's English were beyond the system's limited understanding it was the user's responsibility to restate his query. To prevent the negative psychological impact upon a user caused by a system rejecting a query for apparently arbitrary reasons, Codd proposed some improvements to interface design. The user should be presented with a simple data model because his view of the data influences the way he formulates queries. Query formulation should be kept separate from database search until the user and the system agree upon the user's intent. To achieve this, the user's query is translated by a semantic grammar mechanism into a precise internal language, **Alpha**, based on relational calculus. In the translation process an intermediate form, **Inter-Alpha**, is used. The user should be fed back a precise restatement of his query that he may verify the system has understood his request. In the case of any misunderstanding, the user should be engaged by the system in a clarification dialogue. While Codd clearly states the importance of a good **NLI** to databases, **RENDEZVOUS** is limited in its ability to deal with input that doesn't conform to its internal grammar.

The **LIFER** [HEN77] system has been used to build **NLI**'s for a medical database and a computer-based expert system, but the most complex system built with it is **LADDER** [HEN78] which provides natural language access to a large database of U.S. Navy information distributed over different computers across the United States. Users do not need to know where data is stored. Nor do they need to know a special data query language. Instead, they use a subset of English pertinent to the domain of discourse which **LIFER** translates into a general database query. The rest of **LADDER** handles the specifics of the query. The **LIFER** system, also employing semantic grammars, is an improvement over **RENDEZVOUS**. **LIFER** contains a spelling correction feature. It allows language extension through definition of new words and syntactic structures in terms of old. It allows the missing constituents in elliptical inputs to be deduced from previous input. Lastly, it supports interrogation of the underlying language definition through a grammar editor. Thus, language definition and parsing can be intermixed. **LIFER** does have some limitations. Because it is designed to build interfaces that retrieve from (rather than update) databases, it does not handle assertions. Designed for wh-type questions, **LIFER** does not support many yes-no questions. And **LIFER** has trouble with input that displays syntactic or semantic ambiguity. For example, in the request:

Name the ships from American home ports that are within 500 miles of Norfolk

it is not clear whether the relative clause should modify *ships* or *ports.*

The designers of **LSNLIS** [WOO72] acknowledge that theirs is one of the first usable natural language interfaces and as such emphasizes the translation from English into a formal query language while ignoring the problem of input that doesn't follow a strict parse.

The **PLANES** [WAL78] system is a large relational database of aircraft flight and maintenance data. Like **LSNLIS**, it too uses an **ATN** parser, but is more tolerant of nongrammatical requests. It handles ellipsis and several types of pronoun reference, abbreviations, and a variety of syntactic structures including relative clauses. **PLANES** also feeds back to the user a precise representation of its understanding of the user's request. **PLANES** draws upon ideas put forth in the design of **LSNLIS** and **RENDEZVOUS**.

This section has presented the motives of designers of some of the first non-experimental **NLI**'s to database systems. While claiming success, the designers generally note a shortcoming as the problem of handling unexpected input. The success of **NLI**'s can be expected to increase as further techniques are developed to make them more robust.

2.2 Flexible Natural Language Interfaces in CAI Systems

The flexible handling of linguistic phenomena has significant implications within Computer Assisted Instruction (**CAI**) environments. The first subsection presents a pedagogical motive for designing sophisticated natural language interfaces. The second looks at recent work that has been done on improving **CAI** and again reveals a need for further work on natural language understanding.

2.2.1 Language and Learning

The analogy employed by designers of educational computer systems is that of a socratic dialogue between student and tutor (the computer). The student is assumed to have little understanding of some concept while the computer is assumed to have the complete understanding of an expert. Through dialogue, the computer tutor aids the student in acquiring knowledge. But in order for this to happen, the communication must be flexible.

Studies have firmly shown the impact communication has on learning [BAR69, BAR75, BUL75, DOU79]. Authors writing about **CAI** courseware [BOR80, NIE80] in particular have noted that **CAI** dialogue is similar to any communication; consequently, one must consider what is communicated, to whom, and how. Learners are not merely passive recipients of knowledge, and if computer tutorials are to achieve higher educational goals than one of rote learning, they must employ the skills of effective educational dialogue.

> As the form of communication changes, so will the form of what is learnt. One kind of communication will encourage the memorizing of details, another will encourage pupils to reason about the evidence. . . . From the communication, they will also learn what is expected of them as pupils, . . . whether they are expected to have ideas of their own or only remember what they have been told, . . . to take part in the formulating of knowledge, or . . . to act mainly as receivers.
>
> Douglas Barnes
> *From Communication to Curriculum*, p 15.

Concept learning involves processes of accommodation and assimilation.[5] Briefly, new knowledge must be assimilated by a learner in terms of what he already knows. Sometimes new knowledge conflicts with a learner's world view in which case old knowledge must be restructured to accommodate the new. Unfortunately, we tend to regard knowledge as existing independently of someone

[5] Piagetian learning theory has been directly applied to computerized education environments. For a discussion of this, the reader is referred to Seymour Papert's *Mindstorms* [PAP80] in which the foundations of the LOGO system are presented.

who knows when, in fact, it must be brought to life afresh within every knower by his own efforts—efforts primarily involving language.

Educational theorists have noted that if we consider language as a means of learning then we are regarding the learner as an active participant in the making of meaning [BAR75]. Higher processes of thinking are achieved by the interaction of language behaviour with other mental and perceptual powers [BUL75]. Language is a continuous heuristic performed upon our experience of the world in an effort to make it meaningful [DOU79].

In the effort of acquiring and restructuring knowledge, a student will use free form language: false starts, broken off utterances, anaphoric references, pronominalizations, and so on. Burton [BUR79] argues in favour of **NLI**'s in computer tutoring systems. The student must be free to concentrate on the task at hand. Brown et al. [BRO82] note the importance of a system's ability to recognize alternate wordings of the same concept. But to go beyond this, a computer tutor should be equipped with some means of interpreting less than perfectly formed utterances which a student, ignorant of a concept, may be incapable of making.

Significant progress was made when **CAI** authors realized that concept learning cannot be done by rote, that tutoring is needed to promote understanding. The success of **CAI** systems will increase as advances are made in the flexibility of the tutorial dialogue.

2.2.2 Intelligent CAI

Recently, significant advances have been made in **CAI** systems, and flexible **NLI**'s are important even to different areas of research concentration.

Early **CAI** systems were designed at best—sometimes they were simply "electronic page turners"—as drill and practice monitors presenting problems selected at a level of difficulty appropriate to an individual student's performance. For this reason, such systems were termed *adaptive*. Because of the simplicity of the task domain, models of the student could be based on parametric records of performance rather than explicit representation of knowledge.

A driving goal behind the application of Artificial Intelligence (**AI**) techniques to **CAI** was to extend both the task domains and the adaptiveness of earlier systems. Some of the first *intelligent* **CAI** (**ICAI**) systems were termed *generative* for their ability to generate problems from a database representing a particular subject. But work went beyond this to create *reactive learning environments* [BRO75, BRO82] with the student actively engaged by the instructional system in a tutorial dialogue guided by the student's interests and misunderstandings. Recently, research has focused on facilitating *learning by doing* to allow students to gain experiential knowledge through application of factual knowledge. **ICAI** attempts to transform a student's misconceptions into constructive learning experiences. **ICAI** systems have been developed to tutor various subjects, to create student-initiated learning environments, and to assist diagnosis and assessment.[6] Mechanisms are being developed to analyze student learning behaviour and to employ effective tutoring strategies, both of these in terms of skills that should be learned [BARR77, BRO78, BUR82]. For this to be possible, a system must have extensive knowledge and problem solving expertise, student modeling and diagnostic capabilities, and a sophisticated tutoring and explanation mechanism.

Thus, an **ICAI** system can be seen as composed of three components [BARR82, COL85]: an expertise module, a student model, and a tutoring mechanism. The first component contains information on a particular subject or

[6] For a survey of **ICAI** see [BARR82] and for detailed discussion see [SLE82].

on problem solving skills relevant to that subject. The application specific knowledge and inference mechanisms of this component resemble the expert systems that have been developed for such areas as chemistry, medicine, and geology. The second component must model not only the student's understanding, but misconceptions and difficulties as well. This information has to be inferred from the student's answers and problem solving behaviour. In addition to the student model, this component includes diagnostic algorithms to determine the student's unmastered skills. The third component must make decisions about what to teach and how. The system should be able to assess the process by which a student derives his answers and then make judgements about where he may be going astray in order that it may provide adequate help. It is this component that most directly communicates with the student.

There are so many facets to overall learning systems that researchers necessarily focus on certain aspects while ignoring others. **AI** applications to **CAI** include natural language understanding, knowledge representation, inference methods, and such specific applications as electronics trouble-shooting and medical diagnosis. Nevertheless, despite the necessity of limiting the concentration of research, flexible natural language capability is often an important consideration. Handling ill-formed input is important to all three components of **ICAI** systems.

It is the expert component's task to generate problems and evaluate the student's solutions. One area that has been investigated is that of special purpose inference techniques. The main pedagogical motive behind the **SOPHIE** systems [BRO75, BRO82] is that of experiential learning. The student is engaged in a problem solving process giving rise to experiences that structure factual knowledge. An extension of mixed initiative student-computer dialogue, **SOPHIE** is a *reactive learning environment.* Students learn from their mistakes. A tutoring system should, therefore, allow an interactive one-to-one relationship between student and expert wherein the student can experiment with hypotheses during problem solving and receive feedback and criticism of his ideas. Moreover, the student should be able to ask questions of the expert. Clearly, this requires that the student be able to communicate his ideas to the machine and that a dialogue mechanism robustly handle the constructs that arise in conversation.

It is the student model component's task to represent the student's understanding of a subject and perhaps diagnose the underlying cause of error in some procedural skill. Thus, another area of investigation has been that of creating a model of the student from his observable behaviour and determining what subskills he has not mastered. A system to diagnose errors in a procedural skill must distinguish between goals and methods of achieving those goals, and it must represent both the correct methods of achieving goals and the incorrect. Given this, the diagnosis capable of such a system is determining what set of incorrect methods, or perturbations of correct methods, a student has employed to obtain his results.

A system currently exists to diagnose procedural errors in Mathematics. The **DEBUGGY** system [BRO78, BUR82] examines a student's answers to subtraction problems and attempts to deduce how a student's algorithmic behaviour differs from the correct procedural skill. At the heart of the system is the **BUGGY** model. A student's knowledge cannot be represented just as a subset of an expert's because misconceptions are not a subset of correct skills. Therefore, Burton and Brown posit the idea of a perturbation construct: misconceptions are to be represented as variants of correct skills.

In determining a student's arithmetic misconceptions, **BUGGY** operates under the assumption that errors are not random but are instead modifications of correct procedures. An attempt is made to determine which internal incorrect rules contained in the **BUGGY** model give results equal to the student's answers to subtraction problems; that is, the system tries to predict the student's responses.

BUGGY's knowledge base includes representations of about one hundred and ten primitive arithmetic procedural errors. The results of applying these to subtraction questions are compared to the student's answers. The system selects those bugs that account for at least one wrong answer. Heuristic devices are then employed to reduce this set. For example, procedural errors that are subsumed by others are removed, or some errors may be combined to form compounds. After errors have been iteratively removed or combined, the remaining ones are classified according to how well they explain the student's answers and from these the system tries to pick one as the best explanation.

While student modeling and misconception diagnosis in a domain such as Mathematics may not require an elaborate **NLU** mechanism, one might conceive of a diagnostic system for sentence misconstructions that does. Poor understanding of English usage and basic sentence construction is quite common [BAK81]. A study done in secondary schools [DIE74] shows that the most frequent errors can be classified into about twenty categories. One can find underlying causes of misunderstanding and suggestions for teaching correct usage [GAT80, SHAU77, WEA79]. An **ICAI** system to diagnose sentence construction would be a useful educational tool and an interesting area of investigation. **NLU** systems tend to focus only on understanding ill-formed input, not determining the cause of error. Educational diagnosis systems, on the other hand, while focusing on the cause of error, have not been applied to language. An expert system to diagnose sentence misconstructions must attempt to tie the two together.

It is the tutor component's task to integrate curriculum, teaching methodology, and dialogue. Research here is varied. It includes problem selection, performance monitoring, and remedial material selection. It includes issues such as whether the system should debug the student's errors or the student be encouraged to debug his own and issues such as whether coaching or mixed-initiative is a better strategy. The **BIP** system [BARR77], for example, employs an *adaptive instructional strategy* wherein the sequence of instructional actions are a function of the student's performance. Different areas of instruction require different approaches to individualizing the tutorial. Some areas, such as those requiring memorization, are describable as a linear Markov process, but this is not so of others where facts must be acquired and integrated. **BIP**, therefore, describes each problem in terms of the skills it develops, builds a model of the student's state of knowledge, and makes tutorial branching decisions on the basis of a simple success-fail history. The aspect of this component relevant to this paper is its direct communication with the student.

While **ICAI** designers have had success with their systems, there are some commonly acknowledged shortcomings:

- Systems assume particular conceptualizations hence force a student's performance into this framework. Unable to work within a student's conceptual framework, these systems cannot diagnose misconceptions.
- Interaction is too constrained. A student's expressiveness is limited; consequently, so is the tutor's diagnostic mechanism.

As discussed earlier, concept formation and communication are interrelated and the fact that shortcomings have been identified by the **ICAI** designers suggests a need to further artificial intelligence techniques to enhance robustness and responsiveness. Indeed, in the introduction to their survey of the most sophisticated **ICAI** systems, Sleeman and Brown [SLE82] identify as one of the areas of continuing research the implementation of friendly interfaces and conversational systems.

Summary

To conclude this chapter, let us recapitulate some of the requirements of a flexible natural language interface:

- The user must be able to obtain information without technical knowledge of computers and with a minimum of training.
- The user must be free from consideration of a constrained interface, free to concentrate on the task at hand in his native language. This means keeping to a minimum both the amount of information a user must make explicit in the words he chooses and the number of words he enters.
- The system should present a consistent model of its capabilities with its conversational ability at a level of sophistication equal to that of the type of question it can answer.
- The system should employ clarifying dialogue to feed back its understanding of a user's request and to ask questions about constituents it doesn't understand.
- The user should be able to query the system both with questions about the knowledge base and with *metaquestions* about information and the language definition.
- The user should be able to add new words and syntactic structures to the knowledge base.
- The system should recognize alternate wordings of the same concept.
- The system should tolerate errors of spelling and grammar and suggest corrections wherever possible.
- The system should recognize complex syntactic constructions including abbreviations, context dependent anaphoric references, ellipses, pronominalizations, relative clauses, and incomplete sentences.

Many of these requirements may be realized through a robust parsing mechanism that accepts a wide range of input. This will be discussed in the next chapter.

Chapter 3 Extragrammaticality in Natural Language Interfaces

The last chapter discussed the importance of developing flexible natural language interfaces. One of the major stumbling blocks has been how to handle input that is not strictly *correct*, input that may be called *extragrammatical*.

3.1 What is Meant by Extragrammaticality ?

Why call input *extragrammatical* rather than *ungrammatical*? A sentence is considered extragrammatical if it cannot be accounted for—if it is viewed as ill-formed—by a particular grammar.

A grammar of a language is a model of the linguistic competence of a user of the language [CHO65, RAD81]. There are two types of linguistic competence, pragmatic and grammatical, but the former will not be pursued in this paper.[7] Grammatical competence subsumes three types of linguistic ability: syntactic, semantic, and phonological. Phonology is not of concern here as it pertains to spoken language. Semantics is important to NLU systems such as database front-ends where, for example, English sentences are converted into an internal logical representation for querying. Syntactic competence has two aspects: judgment of well-formedness, and judgment of structure. Our intuitions about well-formedness tell us a sentence like:

John likes fast cars

is syntactically correct and our intuitions about structure tell us that *fast* modifies *cars* and not *likes*. It is these sorts of abilities that a grammar attempts

[7] Pragmatic competence involves language as it is employed in conversation and is often studied in the area of discourse analysis, but it may be embodied in certain NLU systems that use scripts [BARR81, RIC83].

to model.

Now, if a sentence like the one above is grammatical or well-formed, what type of sentence is considered ill-formed? The notion of ill-formedness is by no means clear cut and one must take care to specify what aspects of it are being considered.

It is necessary to distinguish between descriptive well-formedness and prescriptive correctness; that is, sentences like:

I am bigger than what you are

cannot arbitrarily be called incorrect because in some dialects they are perfectly well-formed. However, problems of idiolects, dialects, and sociolects are better left to the study of sociolinguistics.

Another problem with ill-formedness is deciding what might be wrong with a sentence that sounds odd. Are we to call a phrase like:

The tree who we saw

ill-formed? Granted, it is a pragmatic oddity taken on its own, but in the context of a story wherein plants are animate, the human-like qualities implied by the relative pronoun *who* might be quite acceptable.

Radford notes that even ignoring pragmatic circumstances that might lead one to accept sentences that appear linguistically ill-formed, there is still the problem of whether a sentence is ill-formed by virtue of its syntax or its semantics [RAD81]. A sentence like:

We respect herself

might be argued incorrect syntactically because *herself* is a third person feminine singular reflexive pronoun disagreeing in person and number with the first person plural nonreflexive pronoun subject *we*. However, a reflexive pronoun like *herself* can appear as the direct object in a sentence like

Mary respects herself

hence, there is no overall syntactic restriction in English against using *herself* as the object of a transitive verb (p. 11). Instead, the sentence in question might be argued incorrect on the semantic grounds that a reflexive pronoun must take its reference from some compatible antecedent. Differences in how sentences like these are viewed reflect differences in the organization of grammatical models.

The conclusion that this discussion leads to is that we cannot consider ill-formedness in an absolute sense. Certainly we would call a sentence like:

The boy eat the apple

ungrammatical because the subject and verb disagree in number but in other cases we cannot make such an easy judgement. It is not always clear whether a sentence is wrong, or, if it is, why. For the sake of generality, we may consider sentences ill-formed only relative to any grammar which attempts to model language.

The notion of *relative ill-formedness* has important implications to the design of NLU systems. People regularly communicate through sentences that are not strictly grammatical, yet NLU systems do not generally attempt to accept input rejected by grammatical processing. Input may be a syntactically invalid but, nevertheless, semantically meaningful construct; it may be a syntactically correct construct simply beyond the capability of some system; or it may be a correct, but incomplete, construct. Kwasny [KWA80] suggest an approach to handling this sort of input is to assume that just as a normative grammar describing the structure of well-formed inputs can be specified, so can the manner in which input may deviate be specified. This gives an NLU system the appearance of of allowing a wider range of acceptable sentences when in fact it is the case that sentences significantly close to acceptable ones are noted as deviant and accepted as such. In all of these cases the input is ill-formed only relative to the system and not the user. Hence, the term *extragrammatical*, rather than *ungrammatical* or *incorrect*, is used.

3.2 Where Extragrammaticality Arises

Extragrammatical utterances may be found at different levels of linguistic analysis: lexical, sentential, or dialogue.

Dialogue problems are pragmatic and result from a violation of conversation rules: answering a question with a question, making nonsequitur responses, and so forth. By and large, dialogue problems belong to the area of discourse analysis and will not be pursued here.

Lexical problems are confined to individual words and include misspelling, mistyping, incorrect segmentation, and unknown words.

Sentential problems are based on relationships between words and may be of either a semantic or a syntactic nature. They may arise in a variety of situations. For example, with a natural language data base access system a user may be unwilling to change something that he has already typed, or he may believe that the computer will understand a terse military style input. Contrasting with such conscious grammatical violations, errors in normal written English are unconscious, often arising from failure to grasp grammatical conventions. Semantic problems involve omission of necessary information. Syntactic problems include faulty subject-verb agreement, spurious constituents, word order error, legitimate phrases a parser cannot deal with, broken off utterances, unknown words filling a known grammatical role, run on sentences, fragmentary input, elliptical input, and so on.

3.3 Handling Extragrammatical Phenomena

This section presents some examples of extragrammaticality as defined in the last section. The notion of a grammar as a model of linguistic competence was mentioned earlier. Chomsky speaks of different types of linguistic competence. Similarly, he speaks of different levels of linguistic analysis. Such ideas can be used to classify how different extragrammatical phenomena are to be handled, that is, what level of linguistic analysis and grammatical representation are needed:

- Lexical phenomena - a lexical representation such as a lexicon containing parts of speech, preferred meaning, roots, and so on is required.
- Syntactic phenomena - a lexical and a syntactic representation such as a parse tree are required.
- Semantic phenomena - a lexical, a syntactic, and a semantic representation such as extended first-order logic are required.
- Metalinguistic phenomena - a metalinguistic mechanism such as a grammar editor or knowledge base modifier is required.

Although a detailed discussion is beyond the scope of this paper, it is useful to look at a few phenomena and how they have been handled in various systems.

To deal with unexpected input, most robust parsers employ extensions of existing methods, usually at a syntactic or semantic level.

Some work has been done at the level of words. Lexical disambiguation has been handled with lexicons composed of words and associated semantic information. As might be expected, such a scheme becomes cumbersome with large lexicons but can be improved by ranking the semantic information according to a word's preferred usage. Ambiguities are resolved by using the ranking in conjunction with local contextual information.

Metalinguistic phenomena are simply those that reside at a level above a grammar. They involve the techniques a writer employs in developing a grammar. The **LIFER**, **SAUMER**, and **ProGrammar** systems contain some metalinguistic capabilities.

New Words and Phrases

Features for handling new words and phrases may be found in the **LIFER** and **PLANES** systems. Such a feature may be lexical or syntactic depending on how it is employed.

The first step in the **PLANES** system is to put all individual words into canonical form. Many words are replaced by their root forms and user defined words are replaced with those words for which they are synonyms. A similar process is carried out in **LIFER**.

LIFER is unable to interpret new constructs the first time it sees them; however, the system does allow the user to interactively create personalized syntactic constructs it then will continue to understand. If the system understands some construct *B*, the user can create a new construct, *A*, with a statement of the form *Let A be like B.* In **LIFER**, for example, the user can enter:

Define Bill like William

and the system will continue to treat the two names as synonymous.

LIFER has another feature: paraphrase. With this feature, the user can enter:

Let "Describe John" be a paraphrase of "Print the height, weight, and age of John"

Given that the system recognizes the longer construct, then it would be able to understand requests like:

Describe Mary's sister

To handle input like:

Define "new word" like "old word(s)"

a synonym table may be used with entries made when each new word is defined. Whenever the new word is used again, it is simply replaced by another word or words.

The simple case in which a single word is declared synonymous with an existing word or phrase is a lexical phenomenon. Syntactic analysis is required for something of the form:

Define "new phrase" like "old phrase"

In the example, *Describe John* may be parsed to <imperative verb> <object> and *Print the height, weight, and age of John* to <imperative verb> <noun modifiers> <object>. Since the object, *John*, is the same in both cases it may be dropped leaving a correspondence between *Describe* and *Print the height, weight, and age of.* When an input like *Describe Mary's sister* is entered the full expansion can be got from the synonym table.

Ellipsis

Elliptic utterances are characterized by the omission of some sentential constituent that can be easily subsumed in a particular sentence yet inferred from the context of discourse. Two types of ellipsis may be identified: contextual and telegraphic. Systems equipped for handling contextual ellipsis include **LIFER**, **PLANES**, and **SOPHIE**.

Contextual ellipsis is characterized by the constituent being found in a previous sentence. For example, the phrase:

Tom has

makes little sense in isolation but is appropriate in the context of:

Who has taken my book?
Tom has.

What appears to be a sentence with an incomplete predicate is, nevertheless, acceptable. Similarly, a solitary prepositional phrase:

To the theatre

is an appropriate response to the question:

Where are you going?

Telegraphic ellipsis is characterized by the omission of words that convey little meaning. This occurs when the sentence follows a common form such as a newspaper headline or a sign in a shop:

Three chairs no waiting

Supporting both types of ellipsis in an **NLU** system allows a user to follow a natural tendency to abbreviate. A hypothetical database system might allow:

```
> Who is the president of the company?
> The secretary?
> List profits each item
```

Ellipsis can generally be handled syntactically. Contextual ellipsis can be handled if the utterance replaces a constituent in the parse tree of a previous utterance. For example, the elliptic utterance, *the secretary?*, is parsed as a noun phrase and fitted in as the object in the parse tree of the previous utterance. Now the elliptic sentence can be interpreted as "Who is the secretary of the company?". Although the **SOPHIE** and **LIFER** systems employ semantic grammars, their approaches to handling ellipsis are syntactic.

Consider two consecutive queries that may be presented to **SOPHIE**:

What is the base emitter voltage of Q6?
What about Q3?

When the second query is processed, the appropriate grammar rule will contain uninstantiated placeholders for constituents that depend upon context. The context is provided by a history list of instantiated placeholders and grammar rules used.

The approach taken by **LIFER** is to see if a contiguous set of words is syntactically analogous to a contiguous subset of words in a previous input. The elliptical phrase is then fitted into the parse representation of the complete phrase. However, using analogy patterns derived from parse trees means an elliptical utterance must match exactly some constituent of a previous parse and so lacks generality.

Another approach to ellipsis is found in **PLANES**. The system utilizes **ATN** subnetworks, case frames, and special context registers. The registers are used to supply missing constituents in elided sentences.

Disagreement

There are a number of extragrammatical phenomena involving disagreement among constituents: disagreement in number, case, person, mood, or voice. The sentences:

The two apple are mine
Socrates am mortal

exhibit number and person disagreement respectively. Sentences such as these are close enough to being grammatical that they are perfectly intelligible and should be treated by an **NLU** system as less preferred variations of acceptable sentences.[8]

[8] As a matter of interest, disagreement violations are found in certain dialects of English. Nonstandard usage includes inflected plurals, double negatives, third person

Disagreement may be classified as a syntactic phenomenon. To handle sentences of this category, Kwasny [KWA80, KWA81] employs techniques of test and category relaxation. In terms of an **ATN** parser, test relaxation occurs on failure at an arc containing a relaxable predicate. A predicate may be absolutely violable in which case a value of true is substituted for a failed predicate and parsing continues. This would occur with the sentence *The two apple are mine.* Other predicates are conditionally violable in which case an alternate predicate is tried upon failure. Category relaxation expands on Chomsky's hierarchy of categories. To the grammar are added a hierarchy of words, categories, and phrase types. For example, Pronouns may be Demonstrative (this, that...), Personal (he, she...), or Reflexive (yourself, themselves...). In *give he a cookie*, *he* is the incorrect pronoun but since it is found in one of the subcategories of Pronoun, it is accepted.

Summary

Here are a few advantages to designing NLI's that robustly handle extragrammatical input:

- Both the amount of typing and the consequent number of typing mistakes can be reduced.
- A user may choose the level of vocabulary and pronominalization that suits him.
- The user finds an ease in performing similar tasks with fragmentary input interpreted in terms of earlier input.
- The user may extend the range of syntactic structures recognized by the system.
- The user is given freedom in his means of expressing concepts and making queries.

These points and others take on considerable significance in light of the discussion of the last chapter. The material presented in this chapter is far from exhaustive and is itself an area for further research. Nevertheless, it should be

singulars, and so on.

clear that an **NLU** system capable of handling extragrammatical input, one which will accept input beyond that made explicit in its grammar, goes a long way in meeting the requirements of a truly flexible **NLI**.

Chapter 4
Deterministic Parsing

One constant difficulty faced by natural language systems is that the grammar itself defines the language the system accepts. An input sentence may deviate from the accepted language either because the user of the system has made a mistake, or because the grammar itself is wrong or incomplete. The origin of extragrammatical input as we have called it is irrelevant because, whatever the case, a parser is faced with a choice: it must give up, or it must assume the input is reasonable and find a way to deal with something unforeseen by its own rules [KIN83].

Charniak [CHA83] suggests a parser which is "'semi-grammatical' in the sense that it takes a standard 'correct' grammar of English and applies it so long as it can, but will accept sentences which do not fit the grammar, while noting the ways in which the sentences are deviant" (p. 117). A parser which does not check for verb-noun agreement, for example, would not distinguish between:

The fish is dying
The fish are dying

Before an **NLU** system can handle extragrammatical input, what is needed is a parser that enforces grammaticality where possible but behaves gracefully where not. An **ATN** parsing mechanism could not provide this. When a semi-grammatical parser encounters an extragrammatical situation, it must recognize that the input deviates from what is described by the grammar and continue on. A backtracking parser like an **ATN**, on the other hand, when faced with an extragrammaticality, would take this as evidence it had made an incorrect decision and back up to try alternate parses. Not until such a parser has tried unsuccessfully all possible parses of a given sentence does it know there is a problem with the sentence. In other words, at the time it gets stuck, a backtracking parser does not know *why* it has to back up. But with a deterministic parser, failure of rules at a given point may be assumed to be because something is amiss with the input. If a parser is deterministic, it may assume that its input is correct up to the point where it blocks and make a guess at what was intended in order to carry on. Here we turn to the work of Mitchell Marcus.

4.1 Marcus' Deterministic Parser

The theory of parsing put forth by Marcus is an attempt to provide a processing mechanism for current linguistic theory, something linguists themselves have not done [SAM83]. The essence of Marcus' parser is that it provides a model which corresponds to psychological reality by being deterministic. In this important way it is different from the other language processing systems mentioned earlier. It is designed to model how human beings process language—we do not repeatedly try different analyses of a sentence until we find a correct one—rather than provide a tool for machine processing. Consider the sentences:

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

To analyze left-to-right the structure of the above sentences, however, most parsers must simulate nondeterminism, trying one wrong parse, backing up, and trying again. This is the approach taken by ATN parsers. Not until it knows whether there are words after the phrase *sitting in the box* does a parser know if the phrase functions as the complement of the verb *is* or as the modifier of the noun *block*.

Marcus [MARC80] posits a "Determinism Hypothesis":

> . . . 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. (p. 2)

Of course, he does add that "only the *syntactic component* operates strictly deterministically; . . . there is a clear necessity for a strictly deterministic parser to ask questions of semantic-pragmatic components" (p. 3). Following this view, Marcus proposes a parser that never backtracks; instead, it always takes the right path.

Marcus' approach is to parse English with the weakest machine—and within the most restricted framework—possible. This approach might not suffice in the design of a large practical system such as one for translation or question answering: the approach is theoretical, not practical [SAM83]. Instead of presenting a large general grammar, Marcus presents one that captures a small number of complex grammatical phenomena and their interactions.

Marcus discusses his deterministic parser in terms of a a grammar interpreter, **PARSIFAL**, which allows simple rules to capture significant linguistic generalizations: passives, yes-no questions, and imperatives, for example. **PARSIFAL**'s operation is constrained in such a way that to parse sentences which violate grammatical constraints proposed by linguists would require complex, ad hoc grammatical rules.[9]

The operation of the grammar interpreter has some interesting properties. For one, all syntactic substructures created during parsing are permanent. This implies that a backtracking simulation of determinism is impossible. For another, all syntactic substructures created must be output as part of the overall syntactic

[9] Berwick [BER83] notes that a stripped down Marcus parser can be characterized by the LR(k,t) class of grammars. But Nozohoor-Farshi shows this is inadequate. He describes a new class of grammars, LRRL(k), for which deterministic, non-canonical, bottom-up parsers can be derived and shows how grammars parsable by Marcus' system are a subclass of this class [NOZ85a]. He also shows the set of sentences accepted by **PARSIFAL** is a context-free language [NOZ85b].

structure. And this implies that the internal state of the interpreter may have no temporary structures. Further, the parsing process itself has several properties:

- It is partially *data-driven.*
- It can have *expectations* based upon grammatical properties of partial structures already built.
- It has a limited left-to-right *lookahead* facility.

The motivation for these properties can be found in the following sentences:

- Data-driven
  - John went to the store.
  - Did John go to the store?
- Expectations
  - I called $[_{NP}$ John$]$ $[_{S}$ to make Sue feel better$]$.
  - I wanted $[_{S}$ John to make Sue feel better$]$.
- Look-ahead
  - Have $[_{S}$ the boys take the exam today$]$.
  - Have $[_{NP}$ the boys$]$ $[_{VP}$ taken the exam today$]$?

These sentences have some important implications for the parsing process. First, a deterministic parser cannot be strictly top-down. Top-down parsers are hypothesis driven: they choose a goal and try to match the input to that goal. But whether a sentence is a declarative or a yes-no question cannot be decided without examination of the input as the first example above shows. Second, and conversely, a deterministic parser cannot be strictly bottom-up. The second example shows that the phrase *John to make Sue feel better* can be taken as an infinitive complement or as two unrelated constituents. Bottom-up parsers are data-driven: they look at the input and try to drive it towards some goal. A bottom-up parser would fail to make the distinction in the given example. Third, a deterministic parser cannot operate entirely left-to-right. The third example shows that the verb following *the boys* must be examined before the structure of the sentence is known.

Marcus' parser uses two important data structures: a stack of incomplete constituents (partially built syntactic subtrees) called the *active node stack*, and a *buffer* of complete constituents whose higher level function has not been determined. The buffer is a list of five elements of which only a window of three may be accessed at once. These data structures are acted upon by a grammar consisting of pattern-action rules that are partially ordered and partitioned into groups or *packets.* Patterns match elements of the buffer and the top of the stack. The parser attaches buffer elements to the constituent at the top of the stack until that constituent is complete and can be popped from the stack.

Returning to the properties of the parsing process, we see that they are realized through the data structures. Pattern-action rules are triggered by elements of the buffer, thus the parser is partially data-driven. The parser only considers rules belonging to the active packets. Packets are made active to reflect the properties of the constituents in the active node stack. Thus the parser reflects expectations derived from partial structures. Finally, by using a buffer, the parser has a lookahead capability. The elements of the buffer, the lookahead symbols, can be completed constructs as well as bare words. Note that unlimited lookahead would make the notion of determinism vacuous; therefore, Marcus' system uses limited lookahead: no more than three elements can be in the buffer.

Marcus' parser is intended to handle robustly a range of fairly difficult linguistic phenomena and their interactions. The following sections will examine how this is done.

4.2 PARSIFAL's Data Structures

Parse Nodes

Parse nodes represent grammatical constituents, each node being of a given type such as S (sentence), NP (noun phrase), VP (verb phrase), etc. Tree structures of parse nodes represent grammatical structures. Each node has a list of its own *descendents* and is itself attached to its parent. Associated with a node is a set of grammatical *features* summarizing the represented constituent's properties. These are needed to decide upon a node's grammatical role in a larger constituent or upon a constituent's overall grammatical behaviour. For example, the behaviour of a verb phrase is affected by the types of complements a verb takes. The parser builds constituent structures by attaching all subconstituents to the topmost node of that constituent. It must be sure that all attachments are correct because, as already noted, structure building is permanent. Finally, each node has a unique system generated *name*. Figure 4.1 is taken from [MARC80].

```
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 4.1 - Parse tree with features on nodes

The figure is not an exact example of the output produced by **PARSIFAL**; rather, it is intended to show how the system analyses sentences. This is a declarative, major sentence. The subject is a singular, first person noun phrase which dominates a pronoun and is, therefore, not modifiable. The auxiliary verb has future tense and will agree with any singular or plural subject. The object of the verb phrase is a singular noun headed by an indefinite determiner.

Active Node Stack

The parser attempts to add constituents to the top of the stack covering an incomplete constituent with other nodes while building the lower level constituents that are its descendents. Completed, a node is popped from the stack.

The parser may modify two elements of the active node stack: the top node (the *current active node*) and the S or NP node closest to the top (the *dominating cyclic node*).[10] The parser may also examine, but not modify, the descendents of these two: the nodes they dominate. In addition to name, features, and descendents, nodes on the stack have associated with them a list of active rule packets (more on this later).

Buffer

When the parser pops the active node stack, the grammatical role of the completed constituent may be as yet undetermined; that is, the current node may have all its descendents attached but be unattached itself. In this case the node is inserted into the buffer at the left. Of course, other elements of the buffer, inserted at the right, are the unexamined words of an input sentence that are retrieved when an active rule asks about the features of currently empty buffer slots. Thus, each element of the buffer can be a grammatical constituent of any type from a single word to a complete subordinate clause.

Often the parser, to decide what to do with the leftmost buffer constituent, must look at the second or third element. We have seen an example in the last section where the word *have* functions either as an auxiliary, initiating a yes-no question, or as a main verb, initiating an imperative, depending on constituents to its right. Three operations are associated with the buffer: **read**, **insert**, and **delete**. Insertion and deletion are accompanied by right or left shifts to create or fill space.

[10] This is taken from generative grammar theory. S and NP nodes are special in that transformations are applied *cyclically* to the constituents under them. A node which is above another in a parse structure is said to *dominate*.

Operations on the Stack and Buffer

The parser has three fundamental operations:

- **Attach** a constituent to the current active node (stack top).
- **Create** a new active node and push it onto the stack.
- **Drop** a completed node from the stack.

A constituent involved in an **attach** operation may be a newly created node or an element of the buffer. One node is attached to another by being made the rightmost element in its parent's list of descendents. At the same time, if a buffer element is being attached, the node is deleted from the buffer.

A new node is **created** whenever the parser decides the constituents in the buffer actually begin a new constituent. If the parser knows the higher level role of a node at the time of it's creation, it may immediately attach that node to the old current active node. However, sometimes the parser may know that a new higher level constituent is to be begun without knowing its higher level role as when, for instance, it might attach either to the current active node or to some predecessor of that node. We have already seen an example where a constituent would be created without attachment. In the questions:

> Is the block sitting in the box?
> Is the block sitting in the box red?

the verb phrase *sitting in the box* can be attached either as a relative clause to *the block* or as a verb phrase to the main clause itself. Being able to parse a constituent before its grammatical role can be determined is necessary for handling such *nondeterministic* sentences.

Whenever the current active node is completed, it is popped from the stack. A node that was attached upon creation remains attached. However, an unattached node cannot remain in limbo so it is inserted at the front of the buffer at the same time as it is popped off the stack. This is all accomplished by the composite operation: **drop.**

Why Both Stack and Buffer?

Marcus justifies using two data structures on the basis of combined top-down and bottom-up parsing.

Top-down, hypothesis-driven, parsing such as that found in **ATN** mechanisms, logic grammars (see chapter five), or recursive descent algorithms adds subconstituents to a specific node in a parse tree by recursively postulating subconstituents until a terminal symbol is reached that can be checked against the input. The most natural data structure for this is a stack.

Bottom-up, data-driven, parsing attempts recursively to incorporate contiguous sequences of constituents into higher level constituents until a root symbol is reached. The most natural data structure for this is a buffer.

Marcus' parser incorporates both top-down and bottom-up features and so uses two data structures. A node is pushed onto the stack when the parser is looking for its subconstituents. Rather than attempting to find these in a purely top-down fashion, the parser uses its pattern-action rules to recognize, through contiguous sequences in the buffer, subconstituents of the current active node. Constituents may also be recognized bottom-up by rules that are active no matter what the current active node; that is, some constituents may be recognizable no matter what the grammatical environment: for example, a noun phrase. In short, the parser attempts, top-down, to find descendents of the nodes in the active node stack; bottom-up, to find ancestors of the nodes in the buffer.

4.3 Structure and Interpretation of the Grammar

Grammar Rules

Each grammar rule consists of a pattern to be matched against elements of the buffer and the current active node stack, and an action which operates upon those elements. Rules are assigned a priority for arbitration amongst simultaneous matches: the interpreter takes the action of the rule with highest priority whose pattern matches. Before the interpreter will match a rule of a given priority, all higher priority rules must have failed.

Rule patterns are lists of partial descriptions—up to five—to match against each of the three nodes in the buffer as well as the current active node and dominating cyclic node in the stack. Descriptions are tests for grammatical features. Rule actions build constituent structures by:

- creating new parse nodes
- inserting lexical items into the buffer
- attaching a newly created node, or one deleted from the buffer, to the current active node or dominating cyclic node
- popping the current active node from the stack and dropping it into the buffer if it cannot be attached
- assigning features to any of the five accessible nodes
- activating or deactivating rule packets (described below)

Consider, as an example, Marcus' rule to detect the subject-auxiliary inversion that marks a wh question.

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

Figure 4.2 - Grammar rule

The name of this rule is *AUX-INVERSION* and it belongs to the *PARSE-SUBJ* packet. Its pattern tests the first two buffer positions to see if they have the features *auxverb* and *np* respectively. If so, it takes the specified action. The second buffer element is deleted and attached to the current active node. The *PARSE-SUBJ* packet is deactivated and *PARSE-AUX* is activated.

Rule Packets

Rules are organized into *rule packets* which can be activated or deactivated as a group, and each node in the active stack has associated with it at any given time a set of rule packets. The significance of this is that when a node becomes the current active node, the rules in the packets associated with it determine what the system does next. The interpreter only attempts to use rules in active packets because most are applicable only under particular circumstances reflecting global properties of structures already built. Note that several related packets may be active simultaneously. For example, the verb *seems* can take infinitive complements (*It seems to be*) or that-complements (*It seems that*). The packeting mechanism captures most of the left context information about an input sentence but some rules do examine the current active node, the dominating cyclic node, and their descendents. Some of the more important rule packets:

SS-START(Simple-Sentence-START)
: These rules determine the type of a major clause.

PARSE-SUBJ
: These rules pick out and attach the subject of various types of clauses

CPOOL(Clause-POOL)
: These rules are always active whenever any clause-level constituent is being parsed. They are used, for example, to pick out noun phrases.

PARSE-AUX,BUILD-AUX
: These rules initiate building of auxiliaries and attach completed auxiliaries to the dominating S node.

PARSE-VP
: These rules create a VP node and attach the main verb to it. When the VP is complete, a later rule drops it from the stack and attaches it to the main clause node.

SUBJ-VERB
: These rules, involving the deep grammatical relation between the surface subject of the clause and the verb, activate packets to parse verb objects and complements. Some complements depend on the verb of the clause; some on the global properties of the clause.

SS-VP
: These rules attach the verb's objects in major clauses that are not wh questions.

WH-VP,EMBEDDED-S-VP
: These rules parse objects of the verb plus VP-dominated PP's. The

first packet is for clauses with wh heads such as wh questions or relative clauses; the second, similar to SS-VP, for embedded clauses that are neither relative clauses nor indirect questions.

INF-COMP,SUBJ-LESS-INF-COMP,TO-BE-LESS-INF-COMP
: These rules pick up infinitive complements, complements of verbs like *want* that do not require a subject, complements of verbs like *seems* that may take infinitive complements without a preceding *to be*, and so on.

SS-FINAL
: These rules attach clause level modifiers such as prepositional phrases, adverbs, etc. to simple sentences.

EMBEDDED-S-FINAL
: These rules are like those in SS-FINAL except they make a semantic decision whether a modifier is to be attached to the current embedded clause or to be left for later attachment to a higher level constituent.

Example Parse

We may now look at how Marcus' system carries out a parse and will consider the sentence *John has scheduled a meeting.* For a more detailed look at how **PARSIFAL** operates, see [MARC80] or [SAM83].

Every parse begins with a call to *INITIAL-RULE* which creates an S node, pushes it onto the stack, and activates the *CPOOL* and *SS-START* packets.

Amongst the rules belonging to *SS-START* is one, *MAJOR-DECL-S*, whose pattern matches because the first two buffer elements are a noun phrase and verb. The current active node, the S node, is labeled as *declarative* and *major*, *SS-START* is deactivated, and *PARSE-SUBJ* is activated. Figure 4.3 shows the state of the active node stack and buffer after the rule has run.

```
Active Node Stack
S16 (DECL MAJOR S) / (CPOOL PARSE-SUBJ)
Buffer
NP40 (NP NAME NS N3P) : (John)
WORD125 (*HAVE VERB AUXVERB PRES V3S) : (has)
_________

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

Figure 4.3 - After *MAJOR-DECL-S* has run

The parse node on the active node stack has the system generated name *s16*, the features *decl*, *major*, and *s*, no descendents, and the associated active rule packets *CPOOL* and *PARSE-SUBJ*. Before the rule was run, the buffer was empty. Because the rule asked about features of empty buffer slots, a slot filling mechanism was triggered. The parse node in the second buffer position has not been examined by any grammar rule; therefore, its name is simply *word125*, its features are those obtained from the word's entry in a lexicon, and its only descendent is the word *has*. The node in the first buffer position *has* been examined and represents a fully parsed noun phrase, but we shall defer discussion of this.

One of the rules in the *PARSE-SUBJ* packet, *UNMARKED-ORDER*, matches next. It attaches the first buffer constituent to the current active node, deactivates *PARSE-SUBJ*, and activates *PARSE-AUX*: the subject of the sentence has been found and the parser will now look for an auxiliary verb. The details of auxiliary parsing need not concern us here. Briefly, what happens is that the verb *has* is attached to the current active node as a descendent labeled *auxiliary*, the *PARSE-AUX* packet is deactivated and *PARSE-VP* is activated.

```
Active Node Stack
S16 (DECL MAJOR S) / (CPOOL PARSE-VP)
     NP : (John)
     AUX : (has)
Buffer
WORD126 (*SCHEDULE COMP-OBJ VERB INF-OBJ VSPL PAST) : (scheduled)
```

Figure 4.4 - After the auxiliary has been parsed

Note that there is a node in the first buffer position. This is because, in parsing the auxiliary, **PARSIFAL** had to look ahead to see if it was comprised of more than one word as would have been the case in, for instance, ***has been.***

The next rule to match is *MAIN-VERB* which creates a new VP node (making it the current active node), pushes it onto the stack, and attaches the main verb to it. The rule also examines the features of the verb to decide which packets to activate to parse the verb's complements. Only one complement-initiating packet, *INF-COMP*, is made active since *schedule* can take an infinitive complement as in *Schedule the minister to give a talk.*

```
Active Node Stack
S16 (DECL MAJOR S) / (CPOOL SS-FINAL)
      NP : (John)
      AUX : (has)
VP14 (VP) / (SS-VP INF-COMP CPOOL)
      VERB : (scheduled)
Buffer
NP41 (NS INDEF DET NP) : (a meeting)
WORD133 (*. FINALPUNC PUNC) : (.)
```

Figure 4.5 - After *MAIN-VERB* has run

The next rule to match is *OBJECTS*. The state of the system after it has run is indicated in figure 4.6.

```
Active Node Stack
S16 (DECL MAJOR S) / (CPOOL SS-FINAL)
      NP : (John)
      AUX : (has)
VP14 (VP) / (SS-VP INF-COMP CPOOL)
      VERB : (scheduled)
      NP : (a meeting)
Buffer
WORD133 (*. FINALPUNC PUNC) : (.)
___________

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

Figure 4.6 - After *OBJECTS* has run

The completion of the parse is simple. The default rule in *SS-VP*, *VP-DONE*, runs. It pops the VP node from the active node stack and attaches it as a descendent of the S node which has once again become the current active node. This makes the packet *SS-FINAL* active. It, too, contains a default rule, *SS-DONE*, which runs because there are no more constituents in the buffer except for the final punctuation. The node representing the period is attached to the S node and the parse is complete.

Attention Shifting Rules

Marcus proposes additional rules called attention shifting rules that extend the basic grammar and cause the interpreter to *shift attention*, or move a window, from the first buffer element to a later one if it indicates the beginning of another higher level constituent of some sort. The parser constructs the detected constituent, leaves it in the buffer, and then shifts attention back to the beginning of the buffer.

These special rules enable other rules to treat constituents like noun phrases as somehow *primitive*. To understand the need for them, consider how the system might operate without. A rule in the CPOOL packet could match if any word which can start an NP is in the first buffer slot. This would activate another packet to build the NP and drop it into the buffer. Unfortunately, this isn't general enough. Sometimes an NP must be constructed before its first word reaches the first buffer slot. Moreover, while many words *can* begin an NP, they don't always do so.

To solve this problem, the attention shifting rules cause the parser to shift its attention from the actual start of the buffer to a later buffer cell or *virtual buffer start*. After the constituent that triggered the attention shift is completed, it is dropped into the buffer and the virtual buffer start is discarded. Then higher level rules may run as if the constituent appeared fully formed.

Before the interpreter attempts to match the pattern of any high level rule, it first checks to see if the pattern of any attention shifting rule matches. If so—and here let us suppose the constituent that triggers the attention shifting rule is in the *n*th buffer position—it shifts the virtual buffer start to the *n*th cell and runs just the attention shifting rules until the complete constituent is parsed.

Marcus' attention shifting mechanism supports his "Determinism Hypothesis". Consider the **ATN** PUSH arc which is also used to parse subordinate constituents. That it may or may not succeed means it encodes the top-down hypothesis that a constituent of a given type exists at a particular point in the input. Whether the edge of a noun phrase, for instance, is clearly indicated or not, a purely hypothesis-driven parser must hypothesize the existence of such a constituent at every point at which it *could* occur. The attention shifting rules, on the other hand, are data-driven. They allow Marcus' parser to perform syntactic processing that combines expectation-driven and data-driven methods, and to take advantage of guides which are encoded in the input itself.

Buffer Handling with Attention Shifting

To accommodate the attention shifting rules, the index given to the routines **read**, **insert**, and **delete** refers not to the *i*th actual buffer cell, rather, to the *i*th cell from the virtual buffer start which is computed as an offset from the actual start of the buffer. The command **offset**(j) adds j to the previous offset (initially zero) and pushes the result on a stack of offsets. One consequence of this is that buffer elements to the left of the virtual buffer start are invisible. An attention shift is dismissed by the command **pop_offset**.

The reason for keeping a stack of offsets is that there may be attention shifts within attention shifts. Consider the phrases:

a hundred rocks
a hundred pound rock

In the first case, *a* is part of the number phrase *a hundred*; in the second, it acts as a determiner. The third constituent in the buffer must be examined to determine the role of the first. This means the number phrase must be constructed before its leading edge reaches the front of the buffer, and this is accomplished by the attention shifting rules. However, this happens within the parsing of an NP which itself triggered an attention shift.

The provision of attention shifting naturally implies the constituent buffer must be more than just three elements long. Marcus makes two observations: there are no grammar rules that match a constituent in the third cell and that must be constructed by attention shifting rules; and nested attention shifts do not result in even three shifts of the virtual buffer start. He, therefore, limits the buffer to five cells and views the mechanism as a window of three cells sliding in five.

Example Attention Shift

We look briefly at how **PARSIFAL**'s attention shifting mechanism works. Suppose the system is at the point in the parse of *John has scheduled a meeting* where the the rule *OBJECTS* is about to be run. Before the interpreter attempts to match the pattern of *OBJECTS*—which looks at the first buffer position to see if it contains a noun phrase—it first tries the patterns of any active attention shifting rules. The packet *CPOOL* contains one such rule: *STARTNP.*

```
{AS RULE STARTNP IN CPOOL
[=ngstart] -->
Create a new np node.
If 1st is det then activate parse-det.
Activate npool.}
```

Figure 4.7 - Attention shifting rule

At the time this rule is tried, the word *a* is in the first buffer position. A determiner can start a noun group and so has the feature *ngstart.* Because the attention shifting rule triggers, the interpreter shifts attention to the cell occupied by the constituent that triggered it. It does this by executing the command

offset(0) since the constituent is zero positions over from the current virtual buffer start.[11] Figure 4.8 shows the state of the system after the rule which parses determiners has run.

```
Active Node Stack
S16 (DECL MAJOR S) / (CPOOL SS-FINAL)
      NP : (John)
      AUX : (has)
VP14 (VP) / (SS-VP INF-COMP CPOOL)
      VERB : (scheduled)
NP41 (INDEF DET NP) / (PARSE-NOUN NPOOL)
      DET : (a)
Buffer
WORD128 (*MEETING NGSTART NOUN NS) : (meeting)
____________

{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-noun.}
```

Figure 4.8 - After *DETERMINER* has run

Subsequent rules, similar to those for parsing higher level constituents, finish parsing the noun phrase. The attention shift is dismissed by the command **pop_offset**. Completed, the NP node is popped from the stack and dropped back into the buffer whence it will be picked up by the rule *OBJECTS*.

4.4 Linguistic Generalizations Captured by PARSIFAL

The last section discussed the structure of **PARSIFAL**'s grammar and how the system parses. This section looks at the scope of linguistic coverage that Marcus intended to capture.

Marcus claims that his parsing technique captures some of the generalizations underlying English grammar and that the structure of the

[11] Were the parser looking at a phrase like *is a meeting*, the interpreter would shift attention by **offset(1)**.

grammar interpreter itself imposes some of the constraints on transformations found in current generative grammar theory.

Features and Traces

The general framework of the grammar is based on the notion of *annotated surface structure.* Marcus borrows from Winograd the idea of "surface structure *annotated by the addition of a set of features* to each node in a parse tree" (p. 90). From Chomsky he borrows the idea of "surface structure annotated by the addition of an element called *trace* to indicate the 'underlying position' of 'shifted' NP's" (p. 90). The purpose is to represent grammatical information for use in subsequent processing.

Features are used to summarize the grammatical properties of a constituent's internal structure so that later syntactic and semantic analysis routines can access them without actually examining that internal structure. Note that functional information is not included in a constituent's feature set because such information is indicated by position in a parse tree.

One example of how Marcus' parser uses information encoded in features has to do with minor movement rules of generative grammar:[12] the parser undoes them. For instance, the inversion of a subject noun phrase and auxiliary verb which mark a yes-no question is undone and a feature is added to the dominating S node to indicate the sentence is a yes-no question.[13]

Traces are used to indicate the position of constituents that have been displaced by transformations from their underlying logical positions. Following current linguistic theory, a trace is essentially an empty noun phrase (a null-deriving non-terminal) in the surface structure of a sentence without descendents

[12] The notion of extraposition has been investigated in [PER81].

[13] Subject-auxiliary inversion is also undone by CHAT-80—see [PER83, WAR82].

but bound to the noun phrase that filled that position at some level of deep structure.[14] In other words, rather than treat a noun phrase as having been shifted from its original place in a sentence's deep structure, Marcus' parser leaves it where it is and puts in a trace instead with a pointer to the surface NP. Examples of using traces include indicating the underlying position of the wh-head of a question or relative clause and indicating the underlying position of the surface subject of a passivized clause. Another important use of traces in the functioning of the interpreter is this: if a trace has been placed in the buffer by a rule, later rules will be unaware that the NP did not actually appear in the input.[15]

Yes-No Questions, Imperatives, and Passives

Section 4.3 showed how **PARSIFAL** handles a simple declarative sentence. Special use of the buffer captures quite simply several linguistic phenomena:

- An element of the buffer other than the first may be removed allowing discontinuous constituents to be reunited. Sometimes a structure intervenes between two parts of one constituent as, for example, in a yes-no question where the subject comes between two parts of a verb cluster.[16]
- Specific lexical items may be inserted into the input stream permitting the same rules to operate on superficially different cases.
- A trace may be inserted into the buffer rather than directly attached to the parse tree.

In parsing yes-no, questions **PARSIFAL** employs only two rules different from those used to parse declaratives. They essentially negate the noun phrase auxiliary inversion and so remove the need for the grammar to use special rules to handle the discontinuity of the verb cluster. The inversion is undone merely by picking out the subject of the clause found in the second buffer cell.

[14] This idea originates in Chomsky's "Extended Standard Theory"—see [RAD81].

[15] The same use of traces can be found in the CHAT-80 system and in example Gapping Grammars in [DAH84a, DAH84b].

[16] The idea here is similar to that behind gapping rules — see [DAH84a , PER81, POP85].

Parsing imperatives and declaratives differs only in one rule. A rule for imperatives inserts into the buffer the word *you*, labels the sentence node as imperative, and activates the rule packet to look for the subject. This puts the parser in the same state as it would be in if given a declarative clause.

Parsing passive constructions involves using traces. A special rule adds the feature *np-preposed* to the sentence node to indicate the sentence has a preposed subject, creates a trace which is bound to the already found subject, and drops the trace into the buffer. A later rule will attach the trace to a verb so flagging the fact that what appears to be the subject of a sentence is in fact the underlying object.

Summary

This chapter has shown that if an NLU system is to handle extragrammatical input it must first detect it as such, and that this requires a deterministic approach to natural language parsing. One such approach—that of Mitchell Marcus—was discussed in some detail. The next chapter motivates the use of logic programming as a tool for developing natural language systems, and then a logic programming implementation of part of Marcus' system is presented. While further, and experimental, implementation is beyond the scope of this thesis, chapter eight briefly mentions how a parser based on Marcus' deterministic approach could handle extragrammatical input.

Chapter 5
Why Choose a Logic Programming Approach?

In the work of Terry Winograd we can find much insight into **NLU** systems. Often theories of a mathematical or logical structure fail to create a holistic model of language understanding. There are four types of knowledge (syntactic, semantic, heuristic, and world) a person will employ in categorizing experience along lines relevant to his his thought processes. These types of knowledge are used in building interconnections in the mind between concepts. Utterances, then, are *programs* that cause operations to be carried out in the hearer's cognitive system—operations which, through reference to concepts and interconnections, lead to understanding. With this view, Winograd designed his **SHRDLU** system [WIN80].

SHRDLU exhibits procedural embedding of knowledge: specific world facts are encoded as procedures to operate on representation structures. Operations are justified not by facts about language but by a correspondence between the representation and the world being described. Winograd notes that this correspondence is not founded upon universal truths, rather, it is mediated through the programmer who builds the representation structures. And in creating these structures corresponding to facts in a particular domain, the programmer is guided by his ideas of what is true in that domain and his perception of the structures that exist in the mind of the user of the system.

We can infer from this that the understanding ability of any **NLU** system is very much dependent upon what is made explicit—there is always a limit to this—in the system by the designer. Winograd admits that an expert system is not a surrogate expert, only an intermediary, and that there always exists a potential for breakdown. A system will fail when the assumptions underlying its specification are not appropriate for some situation in which it is used. How can the user of a system find out what the relevant assumptions are? Perhaps through the provision of a meta-knowledge facility. We can in fact find in some NLU systems the ability to query not only the knowledge base, but the underlying grammar and deductive mechanisms [HEN77, HEN78, PERL82, SAL85].

Winograd goes on to say that only a small amount of human reasoning fits the mold of deductive logic. He comments that word categorization cannot be equated with a finite set of logical predicates, that a word's applicability depends on the purposes of the speaker and hearer. And so he steers away from a logical deductive model of language. However, Winograd notes that these problems are not automatically solved by moving to a procedural representation, and difficulties still exist.

It seems Winograd is talking about understanding language in a very general sense. We may accept the limits of a logical deductive model to represent human reasoning; and the dependency of understanding upon the purposes of speaker and hearer; and even the constraints imposed by the perceptions of system designers, especially if a meta-knowledge facility is available—we may accept all this and still find within database and expert systems, by the fact that they are of limited scope, no reason to reject a logical representation of language. In fact, inspired by recent developments, researchers are again using logic in NLU systems.

Able to describe logical consequences, traditional logic has long been used to represent meaning. Extensions to predicate calculus to represent the truth of presuppositions and the subtleties of natural language quantification have been reported in [DAH79]. Using logic in database design for both data description and query formalism is discussed in [DAH82]. For a time, parsing knowledge, semantic interpretation, and world knowledge had to be represented through different formalisms and linked through interfaces. However, with the development of **Prolog** [CLO81, PER84], programming in logic [DAH83, KOW79, ROB83] is now possible with logic being used throughout as knowledge representation, programming language, data retrieval mechanism, meaning representation, and even parsing mechanism. An important feature of **Prolog** is that it allows natural language processors to be easily built.

Developments in all of these areas were drawn upon in the implementation of experimental natural language database query systems [DAH81]. These have some points in common with earlier systems such as **LSNLIS** and **LADDER**, most notably the translation of English into an internal formal query language and variable-typing to deal with meaning and aid disambiguation during parsing. The idea is to associate a type with each domain and each element of that domain in the knowledge base. Relations are represented as predicates whose arguments are restricted to be elements of specific domains. Certain queries may then be rejected on the basis of domain incompatibility. Logic provides a particularly elegant means for doing this. In fact, logic programming can be viewed as a generalization of relational databases with logic being used for data, query language, and integrity constraints [FUC83].

But while disambiguation through typing is a point in common, the differences are more significant. A logic programming approach using type-checking allows both semantic and syntactic features of natural language to be incorporated into a single formalism without the need of an intermediate sublanguage. For example, a reading of the question

What is the colour of the car [that is] parked down the street?

in which the antecedent of the relative clause is taken as *the colour of the car* would be rejected immediately on the grounds of semantic anomaly because the subject of the verb *park* cannot belong to the *colour* domain. Syntactic and semantic control are further aided by the incorporation of domain specific knowledge into a lexicon containing entries for each word to specify syntactic role and semantic interpretation. Other systems, **LSNLIS** for instance, use several-pass analysis first to map the surface structure onto a Chomsky-type deep structure [CHO65] considering only syntax, and then to perform semantic checks. **RENDEZVOUS**, too, employs an intermediate sublanguage.

In addition to the advantage of automatic parsing done by **Prolog**, a logic programming approach has many of the desirable features of earlier formalisms. **ATN**'s were developed as a means of performing the type of analysis previously only possible through difficult inversions of transformational grammars (**TG**'s). **TG**'s were developed to explain how sentences with very different wordings can have the same meaning while others with similar wordings can have different meanings. Syntactic relationships between sentential constituents are characterized by deep structures enumerated with context-free phrase structure grammar rules. Sentences are generated by transformations applied to deep structures. **ATN**'s solve the problem of *reversing* transformations. They include structure building actions to create syntactic representations and are flexible in the way they do this. The order in which the pieces are put together need not be the order in which they are found. Simulating a non-deterministic machine, an **ATN** is able to reflect the ambiguity inherent in English. Burton [BUR79] states two advantages of semantic grammars as being their ability to characterize the sentences a system *should* handle and their ability to semantically constrain parsing so aiding disambiguation. All of this can be said of logic programming. The ability to include arguments in grammar symbols and procedure calls in production rules allows syntactic and semantic agreement to be enforced and for meaning-structures to be built.

The first work on logic based databases pioneered the way for further research into logic programming as it applies to different aspects of natural language database systems. Language analysis techniques were further investigated in the **CHAT-80** [PER83, WAR82] and **MICROSIAL** [PIQ82] systems. **SHADOW** [HAD84] was explicitly designed to investigate how certain natural language phenomena translate into precise database queries. A bottom-up parsing strategy—contrasting with **Prolog's** normal top-down approach—that allows left-recursive grammar rules may be found in the **BUP** system [MAT83]. Increasingly, **Prolog** based natural language front-ends are being developed for the Japanese Fifth Generation Computer Systems project [MAR84]. It is interesting that both attribute grammars used for compiler writing and generalized phrase structure grammars for linguistic analysis can be seen as variants of the Horn clause subset of logic [FUC83]. Logic grammars have been applied to the specification of data types [ABR84b], the specification of formal languages, the writing of compilers, and even the translation of English into Spanish [DAH81]. Lastly, considerable work has been done on linguistics and the logic programming formalism itself.

Prolog facilitates the writing of logic grammars in which productions are represented as facts and rules of inference, and parsing as a deductive process carried out by **Prolog** itself. Starting with the first logic grammar formalism, Metamorphosis Grammars, developed by Colmerauer in 1975, many new formalisms have developed. Definite Clause Grammars [PER80], included in the implementation of **Prolog** itself, boast ease of implementation. Definite Clause Translation Grammars [ABR84a] exhibit automatic construction of parse trees and internal representation, as do Modifier Structure Grammars which were actually developed to treat coordination problems. Applying grammars such as these to a database **NLI**, it is possible concisely to specify translation into formal query representation, syntactic analysis, and semantic checking all using a single formalism and without concern for implementation details. Another formalism developed to handle a specific linguistic phenomenon, namely that of left extraposition, is the Extraposition Grammar [PER81]. An extension of Extraposition Grammars to allow both left and right extraposition, free word order, and reference to unspecified intermediate substrings has been developed [DAH84a, DAH84b]. Gapping Grammars, as they are known, have most recently been extended as Unrestricted Gapping Grammars [POP85] to allow more concise description of production rules. An excellent summary of logic grammars may be found in [DAH85b] and a look at parser writing through logic programming in [DAH85a].

Summary

The following points summarize the advantages of logic programming in the design of natural language interfaces:

- Truth and quantification represented through logic.
- Declarative grammar representation.
- Parsing concerns handled by language interpreter.
- Single pass syntactic-semantic analysis.
- Programming language, data, query and data retrieval mechanism, parsing rules, semantic representation, and parsing mechanism all represented with the same formalism.

As logic programming continues to grow, so will its use in building **NLI**'s. In the 1970's **ATN**'s and semantic grammars were developed and applied to **NLU** systems. Then work was done on making such systems more robust [CAR83, GRA83, HAY81, JEN83, KWA80, KWA81]. Now logic programming and logic grammars have been developed. Work has been done on extending grammars themselves to describe natural language more easily, to translate natural into formal language, and so on. It would be interesting to see how an **NLI** built through logic programming could be made to handle extragrammatical phenomena.

Chapter 6
A Prolog Implementation of PARSIFAL

This chapter discusses a **Prolog** implementation of a subset of **PARSIFAL**. It is not easy to reimplement a large program that grew without prior definition or to design a specification of its functional behaviour, and this is a recurring problem for those in **AI** who wish to build on existing work [RIT83]. But Marcus has published a reasonably full description of his grammar so it is at least possible to come up with a rough specification of a system from that. Of course, all the support routines to operate on data structures, which Marcus does not discuss, had to be redesigned. Likewise, a lexicon composed of words and their associated features had to be developed as did user interface i/o routines. Here, however, we will assume their existence and concentrate primarily on the grammar notation used in this **Prolog** implementation of Marcus' parser. For implementation details, the source code is provided in the appendices.

PARSIFAL was built of several components, but of concern here is the grammar. It was written in a specification language, **Pidgin**,[17] that resembles English and must be translated into **Lisp** by an interpreter itself written in **Lisp**. While this entails considerable processing overhead, the idea has certain practical interest. In writing grammars for English, it is useful to write in a high level notation; further, one may wish, as part of the grammatical description, to define how the parsing is to be done. Now, **Prolog** lends itself to both these points; and furthermore, Ritchie [RIT83] suggests a simpler function-argument notation could be used to implement **PARSIFAL** without affecting the central ideas. Using **Prolog**, it is possible to rewrite Marcus' grammar rules in a predicate notation run directly by the **Prolog** interpreter.

[17] For a more complete description of **Pidgin** see [MARC80].

Grammar Rules

In the discussion of the grammar structure, it was noted that rules are assigned priorities to control the order in which pattern matching is attempted. In **Prolog** this is unnecessary and can be accomplished by carefully ordering the rules and allowing the **Prolog** interpreter to do the rest. It was also noted that rules are grouped into packets so that only those rules in the currently active packets are even attempted. Using **Prolog**, packeting is captured by giving all rules belonging to the same packet the same predicate name. For identification, each individual rule name is retained as a comment. The format of rules is:

```
/*
<rule name>
*/<packet name> :-
      <pattern>,
      !,
      <action>, !.
```

Note the cut (!) after the pattern and action. **PARSIFAL**, in accordance with Marcus' "Determinism Hypothesis", was designed to operate without backtracking. But the **Prolog** interpreter is a backtracking system. Once a rule's pattern matches, its action is to be taken and the parser is not to come back to this rule; hence, the cut.

Another point: it may seem odd to have both the pattern and action parts of a grammar rule in the body of a **Prolog** rule. **Prolog** is founded upon the Horn Clause class of logic which permits but one clause in the head of a rule. A grammar rule pattern, on the other hand, may be comprised of several goals, so it has to be included in the body. Nevertheless, the placement of the cut retains the logic of, "If this pattern matches, then take the following action."

Rule Patterns

Pidgin contains a number of ways of expressing patterns to be matched in grammatical rules. For example:

```
[* is verb] [=np]
```

which tests the first and second buffer positions for the features ***verb*** and ***np*** may be written in **Prolog** as:

```
has_feature(1,verb), has_feature(2,np), !
```

Similarly, a test of the current active node for the feature ***np-quest***, written as:

```
[**c; =np-quest]
```

becomes:

```
has_feature(can,'np-quest'), !
```

Some patterns cannot be expressed quite so simply. A pattern like:

```
[there is a whcomp and it is not utilized]
```

tests to see if there is a *whcomp* attached to the dominating cyclic S node and if it has been utilized, that is, there has been a trace *np* bound to it. This becomes in **Prolog**:

```
retrieve_dcn(s,(_,SNodeFeatures,_,Descendents)),
find_descendent(whcomp,Descendents,_),
not(member(utilized,SNodeFeatures)), !
```

The predicate **retrieve_dcn** looks back through the active node stack to find the specified dominating cyclic node. **Find_descendent** searches a list of descendents to find a specific one. Success indicates it exists. The anonymous variable is used as the third argument because it is not necessary the descendent itself be returned. Finally, the **member** predicate is used to check if the *whcomp* has been flagged as utilized by looking at the node's list of features.

The top of the active node stack, the current active node, may be examined by using a predicate **peek**. Similarly, a **read** predicate examines elements of the buffer.

Occasionally no test need be done in the pattern portion of a rule:

```
[t]
```

in Marcus' grammar becomes:

```
!
```

in **Prolog**.

Parse Nodes

Parse nodes are represented in **Prolog** as structures. Each structure has no principal functor but does have four components:[18] an atomic name, a list of features, a list of descendents which may be individual words or further structures, and, in the case of nodes on the stack, a list of active packets. In the following figure, the S node has the name *s1*; the features *decl*, *major*, and *s*; the active packets *cpool* and *ss_final*; and three descendents (*np1*, *aux1*, and *vp1*) which are themselves structures.

[18] Strictly speaking, there is a functor—the comma (,)—which acts as an infix operator.

```
(s1, [decl,major,s], [cpool,ss_final],
    [(np1, [name,ns,n3p,not-modifiable,np],
         [(noun,  [*john,ns,n3p,name,noun,propnoun,ngstart],
               john)]),
     (aux1,[perf,modal,vspl,past,aux],
         [(modal, [*should,vspl,verb,auxverb,past,modal],
               should),
          (perf,  [*have,v-3s,verb,auxverb,pres,tnsless],
               have)]),
     (vp1, [vp],
         [(verb,  [*schedule,vspl,verb,past,part,en,comp-obj,inf-obj],
               scheduled),
          (np2,   [def,det,np],
               [(det,   [*the,ns,npl,n3p,det,def,ngstart],
                     the),
                (nbar,  [ns,nbar],
                     [(noun, [*meeting,ns,noun,ngstart],
                           meeting)])])])])
```

Figure 6.1 - Parse node structure

The list of active packets is left out when the node is dropped from the stack into the buffer.

Operations on Parse Nodes

Marcus' system includes basic commands that act upon parse nodes. The command for creating new parse nodes:

```
create a new <type> node
```

becomes:

```
create(<type>)
```

The operation to replace "deleted" items such as the implicit subject *you* in an imperative statement is:

```
insert the word 'you' into the buffer before 1st
```

In Prolog, this is written:

```
make_buffer_node(you,Node),
insert(1,node).
```

The predicate, **make_buffer_node**, takes a word, looks it up in a lexicon, and returns a parse node structure which includes the type *word*, features from the lexicon, and the word itself.

To attach a buffer element to the current active node, Marcus writes:

```
attach <cell> to c as <type>
```

This becomes:

```
attach(<cell>,<type>)
```

As mentioned, each node has a list of descendents. A node is attached to its parent by being made the rightmost element of its parent's list of descendents. When the parser is finished with the current active node, it may pop the node from the active node stack and insert it at the front of the buffer. The operation:

```
drop c
```

is written:

```
drop
```

There are a few special grammar rules in Marcus' system that know at the time of a new node's creation it is to be attached to the current active node upon completion. Rather than dropping the node into the buffer and immediately attaching it, the node may be attached upon creation to its parent so that when it is dropped it remains attached. The operation in Marcus' system for accomplishing this is:

```
attach a new <type> node to c as <type>
```

Because **Prolog** does not support pointers, a parse node cannot sit both atop the active node stack and on its parent's list of descendents. For this reason it is necessary to attach these special nodes upon completion rather than upon their creation. This is accomplished by:

```
drop_and_attach(<type>)
```

Note that doing the attachment at this time should not affect the overall parse. Even using **Pidgin**, a grammar writer must be aware of the attachment of the current active node. Ritchie [RIT83] comments on an apparent problem in Marcus' system: dropping a node which is both current and attached leaves an already attached node in the buffer which some other rule may try again to attach. He suggests eliminating the combined create and attach operation and adopting a style of grammar writing in which a new node is created unattached by one rule at the start of each new constituent, dropped on completion, and attached by some other rule. This is in effect what has been done for this **Prolog** implementation.

Another operation upon parse nodes is to test if a node has a descendent of a given type. To test if the current active node has a *det* descendent, Marcus writes:

```
if there is a det of c
```

In **Prolog**:

```
peek((_,_,_,Descendents)),
find_descendent(det,Descendents), !
```

To test if the *noun* descendent of the first buffer element is a proper noun, Marcus writes:

```
if the noun of 1st is propnoun
```

In **Prolog**:

```
read(1,(_,_,Descendents)),
find_descendent(noun,Descendents,(_,Features,_)),
member(propnoun,Features), !
```

To look for a descendent of either the S or NP dominating cyclic node, Marcus writes:

```
if there is a <type> of s (or np)
```

In **Prolog**:

```
retrieve_dcn(s,(_,_,_,Descendents)),
find_descendent(<type>,Descendents,_), !
```

It is necessary to have a predicate that finds the dominating cyclic node by searching back through the active node stack again because pointers are unavailable in **Prolog**.

There are a number of operations that manipulate the features of nodes. For example, to add features to the current active node, Marcus writes:

```
label c <feature set>
```

to add a feature to the second buffer element:

```
label 2nd <feature>
```

or to label the current active node with the intersection of a given set of features and those associated with the first buffer element:

```
transfer <feature set> from 1st to c
```

In **Prolog** these are written:

```
label([<feature set>])
label(2,<feature>)
transfer([<feature set>])
```

An examination of Marcus' grammar shows commands such as:

```
attach 1st to c as <type>
attach a new <type> node to c as <type>
attach a new <type> node labeled <feature set> to c as <type>
```

Rather than have operations that appear similar but take different numbers of arguments and have different effects, as Marcus does, it seems clearer to use just the primitive operations **create, attach,** and **label.** Ritchie [RIT83] makes this same observation.

At times Marcus uses conditional expressions:

```
if <boolean>
then <complex action 1>
else <complex action 2>
```

A <complex action> can be a single action, a sequence of actions, or another conditional expression. In **Prolog**, both boolean expressions and actions are predicates. **Prolog** has a special operator, ->, which is useful for readability when a set of predicates are intended to represent an if-then-else construct. So we get:

```
<predicate> ->
        <predicate 1>
    ; <predicate 2>
```

Of course, each of these predicates may be a conjunction of predicates separated by commas.

Traces

Marcus defines a trace to be "an NP which has no daughters but which has associated with it a *binding register* which can be set to point to another NP" (p. 96). For a **Prolog** implementation, because there are no pointers, the interpreter must look back through the active node stack to find the controlling NP. It then extracts from that node and its descendents just the words and copies these as an entire phrase into the trace NP node.

Control of Parsing

Next comes the issue of parser control. The parser tries to match the patterns of only those rules that are applicable at any given point. It may use the rules belonging to those packets that are currently active. Two predicates, **activate**([<packet list>]) and **deactivate**([<packet list>]), add to and remove from the current active node's list of active packets. The parser operates through a recursive procedure, **call_packets**, which looks at the list of active packets and calls each element (a **Prolog** predicate) in turn. When the last one completes, the list is examined again. The list may have changed according to whether any rule action included a call to **activate** or **deactivate**. No rule may fail. Even if no patterns match there is always a default clause which may do nothing more than succeed. When no packets remain active, the parse is finished. **Call_packets** is invoked by an **initial_rule** that starts the parse of an entire sentence but it may be invoked subsequently by an attention shifting rule to parse a noun phrase.

Marcus permits rules to determine their own successors, avoiding the pattern matching process, with an action like:

```
run <rule> next
```

This causes the rule's pattern to be overlooked and its action taken. For those grammar rules which are invoked by others in such a manner, it is possible in **Prolog** to have the rule contain the pattern and, in place of the action, a call to a separate goal. Those rules which want to avoid the pattern simply call the goal which represents the grammar rule's action. The decision to do it this way was not save copying rule actions but to retain linguistic generalizations.

Accommodating attention shifting rules is straightforward. Whenever the parser examines the buffer, the routine for doing this first checks to see if any of the attention shifting rules apply. All attention shifting rules have the same predicate name: **as_rule**. Which packet a rule belongs to (for example, *CPOOL*) is indicated by the first argument. If *cpool* is included in the current active node's list of active packets, then all **as_rule** clauses whose first argument is *cpool* are tried.

Example Session

Here is an example of how the **Prolog** version of **PARSIFAL** works. After the **CProlog** interpreter has consulted the source files, the user interface may be invoked by the command *input*. The user is prompted for a sentence, given back its parse, and asked if he wishes to continue.

```
C-Prolog version 1.5
| ?- [startup].
buffer consulted 5856 bytes 2.1 sec.
grammar consulted 15448 bytes 7.96667 sec.
input consulted 968 bytes 0.550005 sec.
lexicon consulted 11420 bytes 4.53334 sec.
nodes consulted 1504 bytes 0.833341 sec.
stack consulted 5104 bytes 2.63334 sec.
sysutils consulted 476 bytes 0.183345 sec.
tokens consulted 3080 bytes 1.73334 sec.
utils consulted 3696 bytes 2.00001 sec.
startup consulted 47552 bytes 22.9167 sec.

yes
| ?- input.

Sentence to parse
  > John should have scheduled the meeting.

s: [decl,major,s]
    np: [name,ns,n3p,not-modifiable,np]
        noun: john
    aux: [perf,modal,vspl,past,aux]
        modal: should
        perf: have
    vp: [vp]
        verb: scheduled
        np: [def,det,np]
            det: the
            nbar: [ns,nbar]
                noun: meeting
    finalpunc: .

Carry on? y/n : n
```

Figure 6.2 - Example session

Chapter 7
Limitations

7.1 Limitations to PARSIFAL

Marcus' **PARSIFAL** system has attracted some attention from others involved in computational linguistics, mainly with respect to the theoretical claims for its relevance to various linguistic phenomena. It is purported to be a deterministic implementation of "Extended Standard Theory" so it is on the psychological claims that attention has been focused.

Marcus claims that **PARSIFAL** parses those sentences "*which a native speaker can analyze without conscious effort*" (p. 204) and that it fails only in cases of psychological complexity, viz., garden path sentences. Briscoe [BRI83] contradicts this. He notes some problems with the design of **PARSIFAL** which allows it to look ahead into a sentence far enough to resolve all temporary ambiguities except those which are garden paths. NP's can be processed in the buffer because their leading edges can be detected. By the same reasoning that allows this, Briscoe says, PP's, too, could be processed in the buffer, and this would permit the parsing of some garden path sentences. Briscoe makes a second point: preprocessing NP's gives **PARSIFAL** infinite lookahead at the word level which translates into delayed processing. But people process language with almost no delay.

PARSIFAL will sometimes fail on sentences other than garden paths and require semantic support of syntax [SPA83]. Actually, Marcus does concede the need for semantic processing at times and allows its interaction by only at the request of the syntactic component. DeJong [DEJ79] comments that **PARSIFAL** is in some ways similar to **SHRDLU** in that neither permits semantic context to help the syntactic parser. On the other hand, **SOPHIE**, he says, uses context but embeds so much domain specific knowledge in its rules as to be inflexible. DeJong proposes the integration of a parser into a system so as to benefit from predictions the system makes.

Sampson [SAM83] casts some doubt as to whether Marcus' system is completely deterministic:

> . . . If 'looking ahead' and 'backtracking' are just two metaphors for the same thing . . . then it may be all that Marcus can claim is that his system is *relatively* deterministic because his lookahead is *limited*. . . . (p. 96)

Yet Marcus insists his lookahead facility is not tantamount to nondeterminism: for him the important point is that his system discards none of the structures it creates. However, even given the definition of determinism in terms of no building of unused structures, Marcus has been challenged as to whether, within a strictly syntactic framework, parsing can be done deterministically.

An interesting drawback of Marcus' system is a direct consequence of its deterministic parsing method. If a prepositional phrase can be attached to a VP node it is; otherwise, it is left to be picked up by clause level rules. This implies that if a PP can serve as a modifier of the object in a verb phrase, it will do so even if it could also serve as a modifier of the entire clause. Thus, **PARSIFAL** would produce only one parse of:

> I saw the man with the telescope

in which *the man* has *the telescope*. It would miss the parse in which *the telescope* is the instrument of seeing.

Marcus claims that **PARSIFAL** does not handle lexical ambiguity but deals instead with structural ambiguity (p. 26). But in light of the example just given, one begins to doubt the strength of this claim. This problem with a rigidly deterministic parsing method is not limited to ambiguous prepositional phrase attachment. Faced with a sentence which is globally ambiguous, for instance:

The old men and women are muttering

Marcus' system would not produce two alternate outputs taking *old* to be an immediate modifier of *men* or of *men and women.*

PARSIFAL was designed to deal mainly with syntactic phenomena but even within this class coverage is not extensive. By Marcus' own admission, it does not handle phenomena that require extensive semantic processing such as conjunction, ellipsis, verb phase deletion, pronominalization, or prepositional phrase attachment. It does not deal with centre embedded sentences like *The mouse the cat chased squeaks.* Berwick [BER83] notes that it does not handle right extraposition and only handles left extraposition through the use of traces. He proposes some extensions to **PARSIFAL** to handle gapping.

Marcus intended **PARSIFAL** to handle robustly a range of fairly difficult linguistic phenomena and their interactions, but here again some doubt has been expressed. Ritchie [RIT83] notes that Marcus' own test grammar relies heavily on a semantic component and case frame handler (not well documented) to make decisions so making it difficult to assess just the syntactic component despite the fact that Marcus discusses his parser in those terms. This, combined with a relatively restricted set of test sentences, Ritchie says, does not substantiate Marcus' "Determinism Hypothesis".

7.2 Limitations to the Current Implementation

As indicated in the preceding chapter, only a subset of **PARSIFAL** has been implemented for this current research. There are a number of reasons for this. For one, more extensive programming is beyond the scope of this thesis. For another, there are parts of **PARSIFAL** not completely relevant to the idea of writing a deterministic parser in **Prolog**.

PARSIFAL, in addition to its syntactic component, has a semantic case frame interpreter. The published summary of the component is very sketchy so it is difficult to draw up a specification for it. Moreover, most of Marcus' theoretical claims relate to just the syntactic component. Thus, the semantic component has been left out of this implementation.

It seems, however, that a practical system cannot completely ignore semantic processing. Consider, for instance, the problem of prepositional phrase attachment. This may well have to be addressed in a database query **NLI** where qualification is important. Prepositional phrase attachment is an interesting problem in that it exemplifies a situation in which the parser must analyze a constituent before its higher level grammatical role can be determined. Consider the sentences:

I saw the man with the red hair
I saw the man with the telescope

The word *with* indicates the start of a prepositional phrase but the parser cannot know immediately whether the phrase attaches as a modifier of the object or of the entire main clause. Once the whole prepositional phrase has been found, a semantic decision must be made as to where to attach it. In **PARSIFAL**, the decision as to what to do with prepositional phrases like those above is left to the rule packet *SS-VP* which is responsible for parsing verbs and complements. One rule, *PP-UNDER-VP-1*, includes a semantic test to see if a PP can be attached a given verb or if the PP should be left to attach later as a general clause modifier.

One simple guideline is that PP's serving as place and time modifiers generally attach to an entire clause while those serving as other cases attach to a verb phrase. This is exemplified by the sentence:

Take out the garbage before 5 o'clock

But there are exceptions. The verb *schedule* can take a time PP as a modifier, for example:

Schedule an appointment for John before 5 o'clock

One method of partially solving the problem is through the use of case frames which work with the annotated surface structure produced by the parser. Case frames contain the predicate/argument relations in a sentence. However, determining what case a phrase fills can be difficult. The most likely reading of:

The judge presented the boy with the prize

is as a paraphrase of:

The judge presented the prize to the boy

But consider:

The judge presented the boy with the prize to the jury

The problem is that prepositions can mark more than one case. For example, *with* marks commitative, instrument, manner, and neutral cases.

The little Marcus does say about case frames is that they consist of four components: a predicate, which is the word associated with the case frame; specifiers, which provide extra information, such as auxiliary verbs or determiners preceding verbs or nouns; cases; and modifiers which are optional, modify an entire case frame, and are case frames themselves. However, he does not discuss how a decision is made as to whether a prepositional phrase is a case or a modifier. In fact, he comments that the general problem of PP attachment requires extremely complex semantic interaction and is not addressed it in his research. Since Marcus does not provide the code for his case frame interpreter and since none of his test sentences exemplify PP attachment, it is not clear that **PARSIFAL** handles the problem.

For this thesis, much of the syntactic component of **PARSIFAL** has been implemented but not all of it. What have been left out are bells and whistles which do not add to the idea of deterministic syntactic parsing: grammar rules for such things as quantifier phrases and numbers.

While chapter five argued for a logic programming approach to building a natural language parser, as it stands, this implementation of **PARSIFAL** does not follow the principles of clean logic programming.

The active node stack, buffer, and input sentence are implemented as facts in the **Prolog** database and are changed by the database modification commands **assert** and **retract**. However, changes could be made to carry all three from rule to rule as logical variables.

One of the ramifications of such a change would be a complete change in the processing mechanism. As in the original **PARSIFAL**, this implementation, too, uses a packeting mechanism to decide which grammar rules to try at any point in a parse. Packets are activated and deactivated by making changes to the current active node using **assert** and **retract**. Processing is controlled by a special goal, **call_packets**, which invokes itself recursively. It looks at the list of active packets associated with the current active node, calls each member of the list (simply a **Prolog** predicate), and then begins again. Since each rule belonging to the same packet has the same predicate name, all of the relevant rules are tried. Because a rule may change the active packet list, parsing progresses.

A better approach would involve having each rule invoke other rules as goals, not only because this is cleaner logic programming but because it would be necessary in order that the stack, buffer, and input be carried as arguments. This could possibly be done with the **setof** predicate being used to make a list of those rules which are applicable at any time. This is essentially what is done with the active packet list but instead the **Prolog** interpreter itself would be keeping track of what is active. A look at any of the papers describing logic grammars will show that the flow of a parse may be controlled by the way grammar rules reference each other. Conceivably, then, Marcus' grammar could be rewritten following such a methodology. In fact, an improvement which has been suggested by Ritchie [RIT83] is to connect the flow of processing to the grammar rules so that explicit packet activation is not needed and some structure building can be handled automatically.

Another way of cleaning up the current implementation might be to use **Concurrent Prolog**. A grammar rule's pattern could be represented by the *guard* of a clause and its action by the *body*. The *commit* operator would replace the sequential **Prolog** cut which follows a rule's pattern. Although **Concurrent Prolog** tries the guards of all clauses with the same head in parallel, only one would commit to its body because of the mutual exclusiveness of the rule patterns; thus, determinism would be retained.

Chapter 8
Conclusions

This thesis has involved a literature survey and some programming. It has shown the importance of designing very flexible interfaces to systems employing natural language understanding. A major step towards that goal is the design of a robust parsing mechanism capable of handling input not completely anticipated by the system's internal grammar. Before a parser can deal with extragrammatical input, it must first enforce grammaticality where it can, and this implies a deterministic approach to natural language parsing. Such an approach may be found in Marcus' **PARSIFAL** system. Following the recent growth logic programming as a tool for developing natural language parsers, this thesis has also presented a **Prolog** implementation of **PARSIFAL.**

An obvious extension of the work would be the handling of extragrammatical input. Some of the features of Marcus' parser already lend themselves to this. At any given point, not all of the grammar rules are tested, in fact, most rules will be irrelevant. The packeting mechanism prevents **PARSIFAL** from even considering more than just a few rules. Therefore, the number of possible reasons for failure to parse is immediately limited. Another thing: the attention shifting rules allow other rules to assume larger constituents such as noun phrases have already been parsed. This might allow a correction mechanism to operate in terms of constituents at a level higher than that of words alone.

Charniak [CHA83] discusses a parser based on Marcus' that handles ungrammatical input. One way that **PARAGRAM** differs from **PARSIFAL** is that rules, rather than being tested in order of priority, are tested in "parallel". Moreover, the result of a test is not a binary decision, rather a *goodness rating.* The rule with the highest rating is the one that runs next. A rating is the sum of values returned by atomic tests. Charniak gives the following example:

Atomic Test	Add if Succeed	Subtract if Fail
category (e.g. np)	4	15
specific word (e.g. to)	6	15
semantics okay	0	8
other (e.g. agreement)	2	15

The idea is that successful tests raise the score while failed ones reduce it greatly. Note that priorities are not necessary to ensure more specific rules run before less specific ones. A more specific rule, because it has more tests, will get a better goodness rating. Now, with respect to extragrammatical input, one rule will still have the highest rating even though none of them exactly matches the input. **PARAGRAM's** ability to parse ungrammatical sentences stems from the parsing mechanism itself. Furthermore, it can tell where a parse has broken down since it is only then that the goodness rating drops below zero. Consider a sentence like:

We is going to do it

After the subject *we* has been parsed, **PARSIFAL** activates the rule packet *PARSE-AUX*. The rule *START-AUX*, which checks the first buffer element to see if it is a verb, could have added to it an agreement test between the subject and the verb. The sentence above would receive a rating of -11 (+4 for successful category test, -15 for failed agreement). This would still be the highest rating of all the rule patterns in the *PARSE-AUX* packet. Parsing could continue as it should and a note could be made that an agreement test had failed. Charniak does admit there are many ungrammatical, yet understandable, constructs which **PARAGRAM** cannot currently handle. Nonetheless, his ideas would be interesting to try, especially in a **Concurrent Prolog** implementation whence rule patterns would be tried in parallel.

Another enhancement that would increase the range of acceptable input would be the inclusion of morphological analysis like that found in **SAUMER** [POP84].

It was mentioned in the last chapter that Marcus' approach to parsing effectively does away with ambiguity by choosing only one of several possible readings. This might restrict its scope of applicability. It might be possible, using **Prolog**, to overcome this problem through judiciously removing the cut from certain rules and allowing backtracking to produce alternate parses. Given an increased range of acceptable input and a more robust parsing mechanism, a final enhancement would be useful indeed. In an NLU system it may be very important the system be able to provide an explanation as to how a particular parse was done, or, if it failed, why it failed. In order to give an explanation, the system would have to record the path it takes to arrive at a solution. Ideas from **ProGrammar** [SAL85] might be useful here. A parser capable of handling extragrammatical input combined with an explanation facility could be applied to several significant areas:

- A student engaged in a sentence construction tutorial could be told why a sentence is incorrect.
- A database or **CAI** user could be aided in eliciting information.
- A grammar developer, given diagnostics for a sentence which is actually correct, would find clues as to what is wrong with the grammar.

References

[ABR84a] Abramson, H., "Definite Clause Translation Grammars", *Proceedings IEEE Logic Programming Symposium,* Atlantic City, 1984, pp 233-240.

[ABR84b] Abramson, H., "Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars", *TR 84-11* Department of Computer Science, University of British Columbia, 1984.

[BAK81] Baker, S., *The Practical Stylist,* New York: Harper & Row, 1981.

[BAR69] Barnes, D., *Language, the Learner, and the School,* Middlesex: Penguin Books, 1969.

[BAR75] Barnes, D., *From Communication to Curriculum,* Middlesex: Penguin Books, 1975.

[BARR77] Barr, A., and R.C. Atkinson, "Adaptive Instructional Strategies", in H. Spada and W.F. Kempf (eds.), *Structural Models of Thinking and Learning,* Bern: Hans Huber, 1977, pp 83-112.

[BARR81] Barr, A., and E.A. Feigenbaum, *The Handbook of Artificial Intelligence,* v1, Los Altos, California: William Kauffman Inc, 1981.

[BARR82] Barr, A., and E.A. Feigenbaum, *The Handbook of Artificial Intelligence,* v2, Los Altos, California: William Kauffman Inc, 1982.

[BER83] Berwick, R.C., "A Deterministic Parser with Broader Coverage", *Proceedings Eighth International Joint Conference on Artificial Intelligence,* Karlsruhe, West Germany, 1983, pp 710-712.

[BRI83] Briscoe, E.J., "Determinism and its Implementation is PARSIFAL", in K. Sparck Jones and Y. Wilks (eds.), *Automatic Natural Language Parsing,* Chichester: Ellis Horwood Ltd., 1983, pp 61-68.

[BOR80] Bork, A., "Preparing Student-Computer Dialogues: Advice to Teachers", in R.P. Taylor (ed.), *The Computer in the School: Tutor, Tool, Tutee,* New York: Teachers College Press, 1980, pp 15-52.

[BRO75] Brown, J.S., R.R. Burton, and A.G. Bell, "SOPHIE: A Step Toward Creating a Reactive Learning Environment", *International Journal of Man-Machine Studies,* v7n5, Sept 1975, pp 675-696.

[BRO78] Brown, J.S., and R.R. Burton, "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills", *Cognitive Science,* 2, 1978, pp 155-192.

[BRO82] Brown, J.S., R.R. Burton, and J. DeKleer, "Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I, II and III", in D. Sleeman and J.S. Brown (eds.), *Intelligent Tutoring Systems,* London: Academic Press, 1982, pp 227-282.

[BUL75] Bullock Committee, *A Language for Life,* London: Her Majesty's Staionery Office, 1975.

[BUR79] Burton, R.R., and J.S. Brown, "Toward a Natural Language Capability for Computer Assisted Instruction", in H.F. O'Neil (ed.), *Procedures for Instructional Systems Development,* New York: Academic Press, 1979, pp 273-313.

[BUR82] Burton, R.R., "Diagnosing Bugs in a Simple Procedural Skill", in D. Sleeman and J.S. Brown (eds.), *Intelligent Tutoring Systems,* London: Academic Press, 1982, pp 157-183.

[CAR83] Carbonell, J.G., and P.J. Hayes, "Recovery Strategies for Parsing Extragrammatical Language", *American Journal of Computational Linguistics,* v9n3-4, July-Dec 1983, pp 123-146.

[CHA83] Charniak, E., "A Parser with Something for Everyone", in M. King (ed.), *Parsing Natural Language,* London: Academic Press, 1983, pp 117-149.

[CHO65] Chomsky, N., *Syntactic Structures,* The Hague: Mouton & Co., 1965.

[CLO81] Clocksin, W.F., and C.S. Mellish, *Programming in Prolog,* Berlin: Springer-Verlag, 1981.

[COL85] Colbourn, M.J., "Applications of Artificial Intelligence Within Education", *International Journal of Computer Mathematics,* to appear April 1985.

[COD74] Codd, E.F., "Seven Steps to Rendezvous with the Casual User", in J.W. Klimbie and K.L. Koffeman (eds.), *Database Management,* Amsterdam: North Holland, 1974, pp 179-200.

[DAH79] Dahl, V., "Quantification in a Three-Valued Logic for Natural Language Question-Answering Systems", *Proceedings Sixth International Joint Conference on Artificial Intelligence,* Tokyo, 1979, pp 182-187.

[DAH81] Dahl, V., "Translating Spanish into Logic Through Logic", *American Journal of Computational Linguistics,* v7n3, July-Dec 1981, pp 147-164.

[DAH82] Dahl, V., "On Database Systems Development Through Logic", *ACM Transactions on Database Systems,* v7n1, March 1982, pp 102-123.

[DAH83] Dahl, V., "On Logic Programming as a Representation of Knowledge", *IEEE Computer,* v16n10,Oct 1983, pp 106-111.

[DAH84a] Dahl, V., and H. Abramson, "On Gapping Grammars", *Proceedings Second International Logic Programming Conference,* Uppsala, 1984, pp 77-88.

[DAH84b] Dahl, V., "More on Gapping Grammars" *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, 1984, pp 669-677.

[DAH85a] Dahl, V., "Hiding Complexity from the Casual Writer of Parsers", in V. Dahl and P. Saint-Dizier (eds.), *Natural Language Understanding,* New York: Elsevier, 1985.

[DAH85b] Dahl, V., "Logic Based Metagrammars for Natural Language Analysis", *TR 85-1,* Computing Science Department, Simon Fraser University, 1985.

[DEJ79] DeJong, G., "Prediction and Substantiation: A New Approach to Natural Language Processing", *Cognitive Science,* v3, 1979, pp 251-273.

[DIE74] Diederich, P.B., *Measuring Growth in English,* Urbana, Illinois: National Council of Teachers of English, 1974.

[DOU79] Doughty, P., "Language for Living", *McGill Journal of Education,* v14n1, 1979, pp 61-69.

[FUC83] Fuchi, K., "The Direction the FGCS Project will Take, *New Generation Computing,* v1n1, 1983, pp 3-9.

[GAT80] Gatherer, W.A., *A Study of English: Learning and Teaching the Language,* London: Heinemann, 1980.

[GRA83] Granger, R.H., "The NOMAD System: Expectation-Based Detection and Correction of Errors During Understanding Syntactically and Semantically Ill-formed Text", *American Journal of Computational Linguistics,* v9n3-4, July-Dec 1983, pp 188-196.

[HAD84] Hadley, R.F., "SHADOW: A Natural Languge Query Analyser", *TR 84-13,* Computing Science Department, Simon Fraser University, 1984.

[HAY81] Hayes, P.J., and G.V. Mouradian, "Flexible Parsing", *American Journal of Computational Linguistics,* v7n4, Oct-Dec 1981, pp 232-242.

[HEN77] Hendrix, G.G., "Human Engineering for Applied Natural Language Processing", *Proceedings Fifth International Joint Conference on Artificial Intelligence,* Cambridge, Mass., pp 183-191.

[HEN78] Hendrix, G.G., E.D. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a Natural Language Interface to Complex Data", *ACM Transactions on Database Systems,* v3n2, June 1978, pp 105-147.

[JEN83] Jensen, K., G.E. Heidorn, L.A. Miller, and Y. Ravin, "Parse Fitting and Prose Fixing: Getting a Hold on Ill-formedness", *American Journal of Computational Linguistics,* v9n3-4, July-Dec 1983, pp 147-160.

[KIN83] King, Margaret (ed.), *Parsing Natural Language,* London: Academic Press, 1983.

[KOW79] Kowalski, R., *Logic for Problem Solving,* New York: Elsevier, 1979.

[KWA80] Kwasny, S.C., *Treatment of Ungrammatical and Extra-Grammatical Phenomena in Natural Language Understanding Systems,* Bloomington, Indiana: Indiana University Linguistics Club, 1980.

[KWA81] Kwasny, S.C., and Norman Sondheimer, "Relaxation Techniques for Parsing Grammatically Ill-formed Input in Natural Language Systems", *American Journal of Computational Linguistics,* v7n2, April-June 1981, pp 99-108.

[MAR84] Marayama, H., and A. Yonezawa, "A Prolog Based Natural Language Front-End System", *New Generation Computing,* v2n1, 1984, pp 91-99.

[MARC80] Marcus, Mitchell P., *A Theory of Syntactic Recognition for Natural Language,* Cambridge: The MIT Press, 1980.

[MAT83] Matsumoto, Y., et. al., "BUP: A Bottom-Up Parser Embedded in Prolog", *New Generation Computing,* v1n2, 1983, pp 145-158.

[NIE80] Nievergelt, J., "A Paradigmatic Introduction to Courseware Design", *IEEE Computer,* v13n9, Sept 1980, pp 7-21.

[NOZ85a] Nozohoor-Farshi, R., "On Formalizations of Marcus' Parser", Department of Computing Science, University of Alberta, 1985.

[NOZ85b] Nozohoor-Farshi, R., "Context-freeness of the Language Accepted by Marcus' Parser", Department of Computing Science, University of Alberta, 1985.

[PAP80] Papert, S., *Mindstorms,* New York: Basic Books, Inc., 1980.

[PER80] Pereira, F.C.N., and D.H.D. Warren, "Definite Clause Grammars for Language Analysis--A Survey of the Formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence,* v13n3, May 1980, pp 231-278.

[PER81] Pereira, F.C.N., "Extraposition Grammars", *American Journal of Computational Linguistics,* v7n4, 1981, pp 243-256.

[PER83] Pereira, F.C.N., *Logic for Natural Language Analysis,* Menlo Park, California: SRI International, 1983.

[PER84] Pereira, F.C.N. (ed.), *C-Prolog User's Manual,* SRI International, Menlo Park, California, 1984.

[PERL82] Pereira, L.M., P. Sabatier, and E. Oliveira, "ORBI: An Expert System for Environmental Resource Evaluation Through Natural Language", *Proceedings First International Logic Programming Conference,* Marseille, Sept 1982, pp 200-209.

[PIQ82] Pique, J.F., and P. Sabatier, "An Informative, Adaptable and Efficient Natural Language Consultable Database System", *1982 European Conference on Artificial Intelligence,* Orsay, July 1982, pp 250-254.

[POP84] Popowich, F., "SAUMER: Sentence Analysis Using MEtaRules", *TR 84-10,* Computing Science Department, Simon Fraser University, 1984.

[POP85] Popowich, F., "Unrestricted Gapping Grammars: Theory, Implementations, and Applications", *M.Sc. Thesis,* Simon Fraser University, 1985.

[RAD81] Radford, A., *Transformational Syntax,* Cambridge: Cambridge University Press, 1981.

[RIC83] Rich, E., *Artificial Intelligence,* New York: McGraw Hill, 1983.

[RIT83] Ritchie, G.D., "The Implementation of a PIDGIN Interpreter", in K. Sparck Jones and Y. Wilks (eds.), *Automatic Natural Language Parsing,* Chichester: Ellis Horwood Ltd., 1983, pp 69-80.

[ROB83] Robinson, J.A., "Logic Programming–Past, Present, and Future", *New Generation Computing,* v1n2, 1983, pp 107-124.

[SAL85] Salim, J.S., "An Expert System Shell for Processing Logic Grammars", *M.Sc. Thesis,* University of British Columbia, May 1985.

[SAM83] Sampson, G., "Deterministic Parsing", in M. King (ed.), *Parsing Natural Language,* London: Academic Press, 1983, pp 91-116.

[SHA83] Shapiro, E., *A Subset of Concurrent Prolog and its Interpreter,* ICOT Technical Report TR-003, February, 1983.

[SHAU77] Shaughnessy, M.P., *Errors and Expectations,* New York: Oxford University Press, 1977.

[SLE82] Sleeman, D., and J.S. Brown (eds.), *Intelligent Tutoring Systems,* London: Academic Press, 1982.

[SPA83] Sparck Jones, K., and Y. Wilks (eds.), *Automatic Natural Language Parsing,* Chichester: Ellis Horwood Ltd., 1983.

[TRE85] Tremblay, J.P., and P.G. Sorenson, *The Theory and Practice of Compiler Writing,* New York: McGraw Hill, 1985.

[WAL78] Waltz, D.L., "An English Language Question-Answering System for a Large Relational Database", *Communications of the ACM,* v21n7, July 1978, pp 526-539.

[WAR82] Warren, D.H.D., and F.C.N. Pereira, "An Efficient Easily Adaptable System for Interpreting Natural Language Queries", *American Journal of Computational Linguistics,* v8n3-4, July-Dec 1982, pp 110-122.

[WEA79] Weaver, C., *Grammar for Teachers,* Urbana, Illinois: National Council of Teachers of English, 1979.

[WIN73] Winograd, T., "A Procedural Model of Language Understanding", in R.C. Schank and K.M. Colby (eds.), *Computer Models of Thought and Language,* San Francisco: W.H. Freeman and Co., 1973, pp 152-186.

[WIN80] Winograd, T., "What Does it Mean to Understand Language?", *Cognitive Science,* v4n3, July-Sept 1980, pp 209-241.

[WOO70] Woods, W.A., "Transition Network Grammars for Natural Language Analysis", *CACM,* v13n10, Oct 1970, pp 591-606.

[WOO72] Woods, W.A., R. Kaplan, and B. Nash-Webber, *The LUNAR Science Natural Language Information System: Final Report,* BBN Rep. No. 2378, Cambridge, Mass.: Bolt, Beranek and Newman, Inc., 1972.

Appendix 1

Source Code

```
/**********************/
/* GRAMMAR  RULES */
/**********************/

/*
This rule creates an S node and activates the packet of rules to decide on a
sentence's type. It also activates the packet containing attention shifting rules that
are always active on the clause level. Any attention shifting rule that matches
always has priority over rules in other packets. A recursive procedure that controls
parsing is started. Finally, the remaining node on the active node stack is popped
and returned. This is the initial S node which now contains the structure
representing the parse tree of the input sentence.
*/

initial_rule(Tree) :-
                !,
                create(s),
                activate([cpool,ss_start]), !,
                call_packets,
                pop(Tree), !.

/****************************************/
/* SS-START Packet -- Initiate major clauses */
/****************************************/

/*
If a clause begins with a wh marker followed by a verb, it is a wh-question.
*/

/*
WH_QUEST
*/ss_start :-
                has_feature(1,wh),has_feature(2,verb),
                !,
                label([quest,'wh-quest',major]),
                read(1,(_,Features,_)),
                is_it_pp_or_np_quest(Features),
                attach(1,whcomp),
                deactivate([ss_start]),
                activate([parse_subj]), !.
is_it_pp_or_np_quest(Features) :-
                member(pp,Features),
                label(['pp-quest']).
is_it_pp_or_np_quest(Features) :-
                member(np,Features),
                label(['np-quest']).
is_it_pp_or_np_quest(_) :- !.

/*
```

```
If a clause begins with an NP followed by a verb, it is a declarative.
*/

/*
MAJOR_DECL_S
*/ss_start :-
                has_feature(1,np),has_feature(2,verb),
                !,
                label([decl,major]),
                deactivate([ss_start]),
                activate([parse_subj]), !.

/*
If a clause begins with an auxiliary verb followed by an NP, it is a yes/no
question.
*/

/*
YES_NO_Q
*/ss_start :-
                has_feature(1,auxverb),has_feature(2,np),
                !,
                yes_no_q_action, !.

yes_no_q_action :-
                label([quest,'yn-quest',major]),
                deactivate([ss_start]),
                activate([parse_subj]), !.

/*
If a clause begins with a tenseless verb, it is an imperative. The implied subject
'you' is inserted.
*/

/*
IMPERATIVE
*/ss_start :-
                has_feature(1,tnsless),
                !,
                imperative_action, !.

imperative_action :-
                label([imper,major]),
                make_buffer_node(you,Node),
                insert(1,Node),
                deactivate([ss_start]),
                activate([parse_subj]), !.

/*
NP_UTTERANCE
*/ss_start :-
                has_feature(1,np),has_feature(2,finalpunc),
                !,
                label(['np-utterance']),
                attach(1,np),
                attach(1,finalpunc),
                deactivate(all), !.

/*
PP_UTTERANCE
*/ss_start :-
                has_feature(1,pp),has_feature(2,finalpunc),
                !,
                label(['pp-utterance']),
                attach(1,pp),
                attach(1,finalpunc),
                deactivate(all), !.

/****************************************/
/* PARSE-SUBJ Packet -- Subject parsing */
/****************************************/

/*
SUBJ_QUEST
*/parse_subj :-
                has_feature(1,verb),has_feature(can,'np-quest'),has_feature(2,np),
                !,
                ((not(has_feature(1,auxverb))
                ; not(has_feature(3,verb))) ->
                  (create(np),
                   label([trace,'not-modifiable']),
                   bind(whcomp),
                   drop_and_attach(np),
                   label(s,[utilized]),
                   deactivate([parse_subj]),
                   activate([parse_aux]))
                ;  aux_inversion_action),
                !.
```

/*
This rule picks out the subject in clauses where an element of the auxiliary appears before the subject.
*/

```
/*
AUX_INVERSION
*/parse_subj :-
                has_feature(1,auxverb),has_feature(2,np),
                !,
                aux_inversion_action, !.

aux_inversion_action :-
                attach(2,np),
                deactivate([parse_subj]),
                activate([parse_aux]), !.
```

/*
This rule picks out the subject in clauses where the subject appears before the

```
verb. This applies to both declaratives and imperatives.
*/

/*
UNMARKED_ORDER
*/parse_subj :-
                has_feature(1,np),has_feature(2,verb),
                !,
                attach(1,np),
                deactivate([parse_subj]),
                activate([parse_aux]), !.

/*****************************************************************/
/* PARSE-AUX and BUILD-AUX Packets -- Rules for building auxiliaries */
/*****************************************************************/

/*
This rule creates a new node to contain the auxiliary construction and indicates its
person/number agreement and tense.
*/

/*
START_AUX
*/parse_aux :-
                has_feature(1,verb),
                !,
                create(aux),
                transfer([vspl,v1s,'v+13s','vpl+2s','v-3s',v3s,
                        pres,past,future,tnsless]),
                activate([cpool,build_aux]), !.

/*
TO_INFINITIVE
*/parse_aux :-
                has_feature(1,'*to'),has_feature(1,auxverb),has_feature(2,tnsless),
                !,
                create(aux),
                label([inf]),
                attach(1,to),
                activate([cpool,build_aux]), !.

/*
Attach a completed auxiliary to the dominating S node.
*/

/*
AUX_ATTACH
*/parse_aux :-
                has_feature(1,aux),
                !,
                attach(1,aux),
                deactivate([parse_aux]),
                activate([parse_vp]), !.
```

```
/* BUILD-AUX Packet */

/*
PERFECTIVE
*/build_aux :-
            has_feature(1,'*have'),has_feature(2,en),
            !,
            attach(1,perf),
            label([perf]), !.

/*
PROGRESSIVE
*/build_aux :-
            has_feature(1,'*be'),has_feature(2,ing),
            !,
            attach(1,prog),
            label([prog]), !.

/*
PASSIVE_AUX
*/build_aux :-
            has_feature(1,'*be'),has_feature(2,en),
            !,
            attach(1,passive),
            label([passive]),
            label(1,[passive]), !.

/*
MODAL
*/build_aux :-
            has_feature(1,modal),has_feature(2,tnsless),
            !,
            attach(1,modal),
            label([modal]), !.

/*
FUTURE
*/build_aux :-
            has_feature(1,'*will'),has_feature(2,tnsless),
            !,
            attach(1,will),
            label([future]), !.

/*
DO_SUPPORT
*/build_aux :-
            has_feature(1,'*do'),has_feature(2,tnsless),
            !,
            attach(1,do), !.

/*
BE_PRED
*/build_aux :-
            has_feature(1,'*be'), not(has_feature(2,part)),
                (has_feature(2,adj) ; has_feature(2,prep)),
                !,
                attach(1,copula),
                label([copula]),
                label(1,[verb,'pred-verb']), !.

/*
AUX_COMPLETE
*/build_aux :-
                !,
                drop, !.

/**************************************************/
/* PARSE-VP and NO-SUBJ Packets -- Verb processing */
/**************************************************/

/*
This rule sets up the state of the S node, creates a VP node and attaches the main
verb to it, and activates the appropriate packets to parse objects and complements.
*/

/*
MAIN_VERB
*/parse_vp :-
                has_feature(1,verb),
                !,
                deactivate([parse_vp]),
                activate_major_or_embedded_final,
                create(vp),
                read(1,(_,VerbFeatures,_)),
                attach(1,verb),
                activate([cpool]),
                check_inf_obj(VerbFeatures),
                check_that_obj(VerbFeatures),
                check_wh_comp,
                check_passive(VerbFeatures), !.

activate_major_or_embedded_final :-
                (has_feature(can,major) ->
                  activate([ss_final])
                ; activate([embedded_s_final])),
                !.

check_inf_obj(VerbFeatures) :-
                member('inf-obj',VerbFeatures) ->
                 (check_to_less_inf_obj(VerbFeatures),
                  check_to_be_less_inf_obj(VerbFeatures),
                  check_subj_less_inf_obj(VerbFeatures),
                  activate([inf_comp]), !)
                ; !.

check_to_less_inf_obj(VerbFeatures) :-
                member('to-less-inf-obj',VerbFeatures) ->
                 (activate([to_less_inf_comp]), !)
```

```
                    ; !.

check_to_be_less_inf_obj(VerbFeatures) :-
                    member('to-be-less-inf-obj',VerbFeatures) ->
                     (activate([to_be_less_inf_comp]), !)
                    ; !.

check_subj_less_inf_obj(VerbFeatures) :-
                    (member('subj-less-inf-obj',VerbFeatures) ->
                      activate([subj_less_inf_comp])
                    ; check_no_subj(VerbFeatures)),
                    !.

check_no_subj(VerbFeatures) :-
                    member('no-subj',VerbFeatures) ->
                     (activate([no_subj]), !)
                    ; !.

check_that_obj(VerbFeatures) :-
                    member('that-obj',VerbFeatures) ->
                     (activate([that_comp]), !)
                    ; !.

check_wh_comp :-
      (wh_comp_not_utilized ->
                      activate([wh_vp])
                    ; end_major_or_embedded),
                    !.

end_major_or_embedded :-
                    (has_feature(s,major) ->
                      activate([ss_vp])
                    ; activate([embedded_s_vp])),
                    !.

check_passive(VerbFeatures) :-
                    member(passive,VerbFeatures) ->
                     (passive_action, !)
                    ; !.

/*
If the main verb is passive, then the dcn S is marked as np-preposed and and a
new trace NP node is created.
*/

/*
PASSIVE
*/passive_action :-
                    !,
                    label(s,['np-preposed']),
                    create(np),
                    label([trace,'not-modifiable']),
                    bind(np),
                    drop, !.
```

```
/* NO-SUBJ Packet */

/*
If an infinitive is encountered and the main verb can be subjectless, this is a
"seems" construction. Note the similarity to the passive case.
*/

/*
SEEMS
*/no_subj :-
              has_feature(1,'*to'),has_feature(2,tnsless),
              !,
              deactivate([no_subj]),
              passive_action, !.
no_subj :- !.

/**********************************/
/* WH-VP Packet -- WH Placement */
/**********************************/

/*
WH_RESOLVED
*/wh_vp :-
              wh_comp_utilized,
              !,
              deactivate([wh_vp]),
              end_major_or_embedded, !.

/*
This rule captures sentences like "What did John give to Mary?"
*/

/*
WH_WITH_PP_NEXT
*/wh_vp :-
              has_feature(1,prep),has_feature(2,np),
              !,
              create_wh_trace_action, !.

/*
This rule captures sentences like "Who did John give the book to?"
*/

/*
WH_WITH_NP_PP_NEXT
*/wh_vp :-
              has_feature(1,np),has_feature(2,prep),
              !,
              objects_action, !.

/*
WH_PP_BUILD
*/cpool :-
              has_feature(1,prep),not(has_feature(2,np)),
              wh_comp_not_utilized,
              !,
              create(pp),
              attach(1,prep),
              create(np),
              label([trace]),
              bind(whcomp),
              label(s,[utilized]),
              drop_and_attach(np),
              drop, !.

/*
CREATE_WH_TRACE
*/wh_vp :-
              !,
              create_wh_trace_action, !.

create_wh_trace_action :-
              create(np),
              label([trace,'not-modifiable']),
              bind(whcomp),
              label(s,[utilized]),
              drop, !.

/*
This predicate succeeds if there is a whcomp attached to the dominating S node
and returns the dcn's list of features so that a check may be made to see if the
whcomp has been utilized.
*/

wh_comp_exists(SNodeFeatures) :-
              retrieve_dcn(s,(_,SNodeFeatures,_,Descendents),_),
              find_descendent(whcomp,Descendents,_), !.

wh_comp_utilized :-
              wh_comp_exists(SNodeFeatures),
              member(utilized,SNodeFeatures), !.

wh_comp_not_utilized :-
              wh_comp_exists(SNodeFeatures),
              not(member(utilized,SNodeFeatures)), !.

/*********************************************/
/* THAT-COMP Packet -- Parse that-complements */
/*********************************************/

/*
THAT_S_START
*/cpool :-
              has_feature(1,comp),has_feature(1,'*that'),
              has_feature(2,np),has_feature(3,verb),
              !,
              create(s),
              label(['comp-s','that-s',sec]),
              attach(1,comp),
              attach(1,np),
              activate([cpool,parse_aux]),
              call_packets,
              drop, !.

/*
THAT_S_START_1
*/that_comp :-
              has_feature(1,np),has_feature(2,verb),
              !,
              create(s),
              label(['comp-s','that-s',sec]),
              attach(1,np),
              deactivate([that_comp]),
              activate([cpool,parse_aux]),
              call_packets,
              drop, !.

that_comp :-
              deactivate([that_comp]), !.

/*
COMP_TO_NP
*/cpool :-
              has_feature(1,'comp-s'),
              !,
              create(np),
              label(['comp-np','not-modifiable']),
              attach(1,s),
              drop, !.

/************************/
/* Infinitive Complements */
/************************/

/*
INF_S_START
*/cpool :-
              has_feature(1,'*for'),has_feature(2,np),has_feature(3,'*to'),
              !,
              create(s),
              label(['comp-s','inf-s',sec]),
              attach(1,comp),
              attach(1,np),
              activate([cpool,parse_aux]),
              call_packets,
              drop, !.

/* INF-COMP Packet */

/*
Note that when this rule matches, so will rules OBJECTS and
OBJ_IN_EMBEDDED_S. To ensure its higher priority, it is called before either of
```

```
the others.
*/

/*
INF_S_START_1
*/inf_comp :-
            has_feature(1,np),has_feature(2,'*to'),
            has_feature(2,auxverb),has_feature(3,tnsless),
            !,
            create(s),
            label([sec,'comp-s','inf-s']),
            attach(1,np),
            activate([cpool,parse_aux]),
            call_packets,
            drop, !.
inf_comp :- !.

/* TO-LESS-INF-COMP Packet */

/*
This rule handles verbs like "help" which take infintive complements with an
implicit "to". Note how, as in the case of imperative sentences, the implied word is
inserted.
*/

/*
INSERT_TO
*/to_less_inf_comp :-
            has_feature(1,np),has_feature(2,tnsless),
            !,
            make_buffer_node(to,Node),
            insert(2,Node),
            deactivate([to_less_inf_comp]), !.

/*
INSERT_TO_1
*/to_less_inf_comp :-
            (has_feature(1,tnsless) ;
             (has_feature(1,np),has_feature(2,finalpunc))),
            !,
            make_buffer_node(to,Node),
            insert(1,Node),
            deactivate([to_less_inf_comp]), !.

to_less_inf_comp :-
            deactivate([to_less_inf_comp]), !.

/* TO-BE-LESS-INF-COMP Packet */

/*
This rule handles verbs like "seems" which take infintive complements with an
implicit "to be".
*/
```

```
/*
INSERT_TO_BE_1
*/to_be_less_inf_comp :-
                (has_feature(1,en) ; has_feature(1,adj)),
                !,
                make_buffer_node(to,ToNode),
                make_buffer_node(be,BeNode),
                insert(1,ToNode),
                insert(2,BeNode),
                deactivate([to_be_less_inf_comp]), !.

to_be_less_inf_comp :-
                deactivate([to_be_less_inf_comp]), !.

/*******************************/
/* Infinitives with Delta Subjects */
/*******************************/

/*
This rule handles verbs like "want" which may have either an explicit or a delta
subject.
*/

/*
CREATE_DELTA_SUBJ_1
*/subj_less_inf_comp :-
                has_feature(1,'*to'),has_feature(1,auxverb),has_feature(2,tnsless),
                !,
                deactivate([subj_less_inf_comp]),
                create(np),
                label([trace,'not-modifiable']),
                drop, !.

subj_less_inf_comp :-
                deactivate([subj_less_inf_comp]), !.

/*********/
/* TIME  */
/*********/

/*
MONDAY
*/as_rule(cpool,BufferCell,Features) :-
                member(noun,Features),member(dow,Features),
                !,
                Offset is BufferCell - 1,
                offset(Offset),
                create(np),
                label([time,dow]),
                attach(1,noun),
                drop,
                pop_offset, !.

/*******************************************/
```

```
/* Pronouns, Proper Names, and Proper Nouns */
/*******************************************/

/*
PRONOUN
*/parse_noun :-
                has_feature(1,pronoun),
                !,
                label(['pron-np','not-modifiable']),
                transfer([ns,npl,n1p,n2p,n3p,wh]),
                read(1,(_,Features,_)),
                is_it_relpron(Features),
                attach(1,pronoun),
                deactivate(all), !.

is_it_relpron(Features) :-
                member(relpron,Features) ->
                  (label(['relpron-np']), !)
                ; !.

/*
PROPNAME
*/as_rule(cpool,BufferCell,Features) :-
                member(name,Features),not(member('not-modifiable',Features)),
                !,
                Offset is BufferCell - 1,
                offset(Offset),
                create(np),
                label([name,ns,n3p,'not-modifiable']),
                activate([build_name]),
                call_packets,
                drop,
                pop_offset, !.

/*
TITLE
*/as_rule(cpool,BufferCell,Features) :-
                member(title,Features),
                !,
                Offset is BufferCell - 1,
                offset(Offset),
                create(np),
                label([name,ns,n3p,'not-modifiable']),
                attach(1,title),
                does_period_follow,
                activate([build_name]),
                call_packets,
                drop,
                pop_offset, !.

does_period_follow :-
                read(1,(_,Features,_)),
                (member('*.',Features) ->
                  (delete(1), !)
                ; !).
/*
NAME
*/build_name :-
                has_feature(1,name),
                !,
                attach(1,noun), !.

/*
END_OF_NAME
*/build_name :-
                !,
                deactivate(all), !.

/*
PROPNOUN
*/as_rule(cpool,BufferCell,Features) :-
                member(propnoun,Features), not(member(name,Features)),
                !,
                Offset is BufferCell - 1,
                offset(Offset),
                create(np),
                label(['propn-np',ns,n3p,'not-modifiable']),
                attach(1,noun),
                drop,
                pop_offset, !.

/**********************/
/* Mainline NP Parsing */
/**********************/

/*
START_NP
*/as_rule(cpool,BufferCell,Features) :-
                member(ngstart,Features),
                !,
                Offset is BufferCell - 1,
                offset(Offset),
                create(np),
                (member(det,Features) ->
                  activate([parse_det])
                ; activate([parse_adj])),
                call_packets,
                drop,
                pop_offset, !.

/* PARSE-DET Packet */

/*
DETERMINER
*/parse_det :-
                has_feature(1,det),
                !,
                label([det]),
                    transfer([indef,def,wh]),
                    attach(1,det),
                    deactivate([parse_det]),
                    activate([parse_adj]), !.

/* PARSE-ADJ Packet */

/*
ADJ
*/parse_adj :-
                    !,
                    (has_feature(1,adj) ->
                      attach(1,adj)
                    ; (has_feature(1,'*,') ->
                      attach(1,comma)
                    ; (deactivate([parse_adj]),
                       activate([parse_noun])))),
                    !.

/* PARSE-NOUN Packet */

/*
NOUN
*/parse_noun :-
                    has_feature(1,noun),
                    !,
                    transfer([time,place]),
                    create(nbar),
                    transfer([time,ns,npl,n1p,n2p,n3p]),
                    attach(1,noun),
                    activate([cpool,nbar_complete]),
                    call_packets,
                    drop, !.

/*
NBAR
*/parse_noun :-
                    has_feature(1,nbar),
                    !,
                    is_proper_noun,
                    attach(1,nbar),
                    deactivate(all), !.

is_proper_noun :-
                    read(1,(_,_,Descendents)),
                    find_descendent(noun,Descendents,(_,Features,_)),
                    (member(propnoun,Features) ->
                     (label(['not-modifiable']), !)
                    ; !).

/***********************/
/* PP Attachment Rules */
/***********************/
```

```
/*
PP
*/cpool :-
            has_feature(1,prep),not(has_feature(1,'pred-verb')),
            has_feature(2,np),
            (not(wh_comp_exists(_)) ; wh_comp_utilized),
            !,
            create(pp),
            attach(1,prep),
            transfer([time,place,wh]),
            attach(1,np),
            drop, !.

/*
OF_PP
*/nbar_complete :-
            has_feature(1,pp),read(1,(_,_,Descendents)),
            find_descendent(prep,Descendents,(_,Features,_)),
            (member('*of',Features) ; has_feature(2,pp)),
            !,
            attach(1,pp), !.

/*
NBAR_DONE
*/nbar_complete :-
            !,
            deactivate(all), !.

/*
These rules decide whether to attach a PP as a modifier of a main verb phrase, an
embedded verb phrase, an embedded sentence, or the main clause. Since the
general problem of PP attachment is semantically complex, the only rules used are
that time modifiers are attached not to a verb phrase but to an entire clause, and
that a PP is attached to the nearest constituent.
*/

/*
PP_UNDER_VP_1
*/ss_vp :-
            has_feature(1,pp),
            !,
            ((not(has_feature(1,time)), not(has_feature(1,place))) ->
              attach(1,pp)
            ; vp_done_action),
            !.

/*
PP_UNDER_VP_2
*/embedded_s_vp :-
            has_feature(1,pp),
            !,
            ((not(has_feature(1,time)), not(has_feature(1,place))) ->
              attach(1,pp)
            ; embedded_vp_done_action),
                !.

/*
PP_UNDER_S_1
*/ss_final :-
                has_feature(1,pp),
                !,
                attach(1,pp), !.

/*
PP_UNDER_S_2
*/embedded_s_final :-
                has_feature(1,pp),
                !,
                attach(1,pp),
                embedded_s_done_action,
                !.

/*********************/
/* Parse simple objects */
/*********************/

/* SS-VP Packet */

/*
OBJECTS
*/ss_vp :-
                has_feature(1,np),
                !,
                objects_action, !.

ss_vp :-
                has_feature(1,'comp-s'),
                !,
                create(np),
                label(['comp-np']),
                attach(1,s),
                drop,
                objects_action, !.

objects_action :-
                attach(1,np), !.

/*
VP_DONE
*/ss_vp :-
                !,
                vp_done_action, !.

vp_done_action :-
                drop_and_attach(vp), !.

/* SS-FINAL Packet */
```

```
/*
S_DONE
*/ss_final :-
                has_feature(1,finalpunc),
                !,
                attach(1,finalpunc),
                deactivate(all), !.

/* EMBEDDED-S-VP Packet */

/*
This rule attaches an object as part of an embedded sentence.
*/

/*
OBJ_IN_EMBEDDED_S
*/embedded_s_vp :-
                has_feature(1,np),
                !,
                attach(1,np), !.

/*
EMBEDDED_VP_DONE
*/embedded_s_vp :-
                !,
                embedded_vp_done_action, !.

embedded_vp_done_action :-
                drop_and_attach(vp), !.

/* EMBEDDED-S-FINAL Packet */

/*
EMBEDDED_S_DONE
*/embedded_s_final :-
                !,
                embedded_s_done_action, !.

embedded_s_done_action :-
                deactivate(all), !.

/*
Even if no rules belonging to this packet match, a call must, nevertheless, succeed.
*/

cpool :- !.
```

```
/************************************************/
/* OPERATIONS ON THE ACTIVE NODE STACK */
/************************************************/
/*
Standard stack operations. Each element is kept as an assertion of the form
active_node_stack( (<stack position>,<node>) ) and the top of the stack is
indicated by the assertion top_of_stack(<top>). Each new stack element has a
position one greater than the previous top of stack.
*/

push(Node) :-
                retract(top_of_stack(T)),
                T1 is T + 1,
                assert(top_of_stack(T1)),
                asserta(active_node_stack((T1,Node))).

pop(Node) :-
                top_of_stack(T1),
                T1 > 0,
                retract(top_of_stack(T1)),
                T is T1 - 1,
                assert(top_of_stack(T)),
                retract(active_node_stack((T1,Node))).
pop(_) :-
                writestring("popping an empty stack"), fail.

peek(Node) :-
                top_of_stack(T),
                active_node_stack((T,Node)), !.

/*
Create a new parse node of the given type and push it onto the active node stack.
Initially, a node has no features (save its type), active packets, or descendents.
*/

create(Type) :-
                conname(Type,NewNodeName),
                push((NewNodeName,[Type],[],[])), !.

/*
Add to the current active node's list of active packets.
*/

activate(NewPackets) :-
                pop((NodeName,Features,OldPackets,Descendents)),
                append(OldPackets,NewPackets,ActivePackets),
                push((NodeName,Features,ActivePackets,Descendents)), !.

/*
Remove from the current active node's list of active packets.
*/

deactivate(all) :-
                pop((NodeName,Features,_,Descendents)),
                push((NodeName,Features,[],Descendents)), !.

deactivate(InactivePackets) :-
                pop((NodeName,Features,OldPackets,Descendents)),
                delete_all(InactivePackets,OldPackets,ActivePackets),
                push((NodeName,Features,ActivePackets,Descendents)), !.

/*
Attach the constituent in the given buffer position as the rightmost descendent of
the current active node, indicate its type, and delete the contents of the buffer
position.
*/

attach(BufferPosition,Type) :-
                read(BufferPosition,(_,DescFeatures,OwnDescendents)),
                pop((ParentName,ParentFeatures,Packets,D1)),
                append(D1,[(Type,DescFeatures,OwnDescendents)],Descendents),
                push((ParentName,ParentFeatures,Packets,Descendents)),
                delete(BufferPosition), !.

/*
Drop an unattached completed constituent from the stack into the buffer. The list
of active packets for this node is no longer needed.
*/

drop :- pop((NodeName,Features,Packets,Descendents)),
                insert(1,(NodeName,Features,Descendents)), !.

/*
Drop a completed constituent from the stack and immediately attach it to the now
current active node. This is used by grammar rules that know for certain the
constituent attaches to the node that immediately dominates it and not possibly to
some higher level constituent.
*/

drop_and_attach(Type) :-
                pop((_,DescFeatures,_,OwnDescendents)),
                pop((ParentName,ParentFeatures,Packets,D1)),
                append(D1,[(Type,DescFeatures,OwnDescendents)],Descendents),
                push((ParentName,ParentFeatures,Packets,Descendents)), !.

/*
Retrieve the dominating cyclic node: S or NP. This is done by searching backwards
through the nodes on the active node stack.
*/

retrieve_dcn(DCN,Node,Pos) :-
                top_of_stack(T),
                find_dcn(DCN,Node,T,Pos), !.

find_dcn(_,_,T,_) :-
                T =< 0,
                writestring("cannot find dominating cyclic node"), !, fail.
find_dcn(DCN,(NodeName,Features,Packets,Descendents),T,T) :-
```

```
                active_node_stack((T,(NodeName,Features,Packets,Descendents))),
                member(DCN,Features), !.
find_dcn(DCN,Node,T1,Pos) :-
                T is T1 - 1,
                find_dcn(DCN,Node,T,Pos).

/*
Find the descendent of the specified type in a node's list of descendents.
*/

find_descendent(Type,[(Type,Features,Descs)|_],(Type,Features,Descs)) :-
                !.
find_descendent(Type,[(_,_,Descs)|_],Descendent) :-
                not(atom(Descs)),
                find_descendent(Type,Descs,Descendent).
find_descendent(Type,[_|Rest],Descendent) :-
                find_descendent(Type,Rest,Descendent).

/*
Bind the current active node, which will be a trace NP node, to the given type of
node which is a descendent of the current dominating cyclic node. Binding
amounts to attaching to the current active node a descendent whose associated
words are the same as those of the node above.
*/

bind(Type) :-
                retrieve_dcn(s,(_,_,_,DCNsDescendents),_),
                find_descendent(Type,DCNsDescendents,(_,_,Descendents)),
                pop((NodeName,Features,Packets,D1)),
                extract_words(Descendents,Words),
                append(D1,[('bound to',[],Words)],D2),
                push((NodeName,Features,Packets,D2)), !.

/*
Extract just the words from a list of descendents.
*/

extract_words(Word,Word) :-
                atom(Word).
extract_words([],[]).
extract_words([(_,_,[])|Descendents],Words) :-
                extract_words(Descendents,Words).
extract_words([(_,_,Word)|Descendents],[Word|Words]) :-
                atom(Word),
                extract_words(Descendents,Words).
extract_words([(_,_,D1)|Descendents],Words) :-
                extract_words(D1,W1),
                extract_words(Descendents,W2),
                append(W1,W2,Words).
extract_words([W1|W2],[W1|W3]) :-
                atom(W1),
                extract_words(W2,W3).

/*
```

```
Create an empty stack.
*/

make_empty_stack :-
                retract_all(active_node_stack(_)),
                assert(active_node_stack((0,([],[],[],[])))),
                retract_all(top_of_stack(_)),
                assert(top_of_stack(0)).

/*
Invoke the rules associated with each currently active packet.
*/

call_packets :-
                peek((_,_,Packets,_)),
                not(empty(Packets)),
                call_each(Packets), !,
                call_packets.
call_packets :- !.

call_each([Rule|Rules]) :- !,
                call(Rule), !,
                call_each(Rules).
call_each(_) :- !.
```

```
/*****************************************************/
/* OPERATIONS ON THE CONSTITUENT BUFFER */
/*****************************************************/

/*
Insert the given contents into buffer position I after first shifting right by one or
two positions to accommodate. The buffer is full if the third cell relative to the
current offset is occupied.
*/

insert(I,Contents) :-
                cell_name(3,RightCellName),
                RightCell =.. [RightCellName,RightContents],
                call(RightCell),
                empty_node(RightContents),
                retract(RightCell),
                J is 3 - I,
                move_right(J),
                cell_name(I,CellName),
                Cell =.. [CellName,Contents],
                assert(Cell), !.
insert(I,Contents) :-
                putback(3),
                insert(I,Contents), !.

move_right(0).
move_right(1) :-
                cell_name(2,SecondCellName),
                SecondCell =.. [SecondCellName,Content2],
                retract(SecondCell),
                cell_name(3,ThirdCellName),
                ThirdCell =.. [ThirdCellName,Content2],
                assert(ThirdCell).
move_right(2) :-
                move_right(1),
                cell_name(1,FirstCellName),
                FirstCell =.. [FirstCellName,Content1],
                retract(FirstCell),
                cell_name(2,SecondCellName),
                SecondCell =.. [SecondCellName,Content1],
                assert(SecondCell).

/*
Delete buffer position I then shift left to fill the vacated cell.
*/

delete(I) :-
                cell_name(I,CellName),
                Cell =.. [CellName,Contents],
                call(Cell),
                not(empty_node(Contents)),
                retract(Cell),
                J is 3-I,
                move_left(J),
                cell_name(3,ThirdCellName),
                EmptyCell =.. [ThirdCellName,([],[],[])],
                assert(EmptyCell), !.
delete(_) :-
                writestring("deleting empty buffer slot"), !, fail.

move_left(0).
move_left(1) :-
                cell_name(3,ThirdCellName),
                ThirdCell =.. [ThirdCellName,Content3],
                retract(ThirdCell),
                cell_name(2,SecondCellName),
                SecondCell =.. [SecondCellName,Content3],
                assert(SecondCell).
move_left(2) :-
                move_left(1),
                cell_name(2,SecondCellName),
                SecondCell =.. [SecondCellName,Content2],
                retract(SecondCell),
                cell_name(1,FirstCellName),
                FirstCell =.. [FirstCellName,Content2],
                assert(FirstCell).

/*
Read the contents of the specified buffer cell. If the cell is empty, it is filled with
the next word in the input list.
*/

read(I,Contents) :-
                cell_name(I,CellName),
                Cell =.. [CellName,CurrentContents],
                call(Cell),
                fill(CellName,CurrentContents,Contents), !.

fill(CellName,CurrentContents,Contents) :-
                empty_node(CurrentContents), !,
                EmptyCell =.. [CellName,CurrentContents],
                retract(EmptyCell),
                retract(input_list([Word|Rest])),
                assert(input_list(Rest)),
                make_buffer_node(Word,Node),
                FullCell =.. [CellName,Node],
                assert(FullCell),
                Contents = Node, !.
fill(_,C,C) :- !.

putback(CellNum) :-
                writestring("warning: putting a word back into the input stream"),
                nl,
                read(CellNum,(_,_,Word)),
                retract(input_list(Words)),
                assert(input_list([Word|Words])),
                delete(CellNum), !.
```

```
/*
Get the atomic cell name of the cell specified by I relative to the current offset in
the buffer.
*/

cell_name(I,CName) :-
                offset_stack([Offset|_]),
                CellNum is I + Offset,
                name(cell,N1),
                integer_name(CellNum,N2),
                append(N1,N2,N),
                name(CName,N), !.

/*
Given an input word, make a parse node for insertion into the buffer.
*/

make_buffer_node(Word,Node) :-
                conname(word,NewNodeName),
                lookup(Word,Features),
                Node = (NewNodeName,Features,Word), !.

/*
Create an empty buffer.
*/

make_empty_buffer :-
                retract_all(cell1(_)),
                retract_all(cell2(_)),
                retract_all(cell3(_)),
                retract_all(cell4(_)),
                retract_all(cell5(_)),
                assert(cell1(([],[],[]))),
                assert(cell2(([],[],[]))),
                assert(cell3(([],[],[]))),
                assert(cell4(([],[],[]))),
                assert(cell5(([],[],[]))).

/*
Start at a zero offset in the constituent buffer.
*/

zero_buffer_offset :-
                retract_all(offset_stack(_)),
                assert(offset_stack([0])).

/*
Push a new offset relative to the current one onto the offset stack. This results in
an attention shift to an effective buffer start to the right of the current buffer start.
*/

offset(New) :-
                retract(offset_stack([Old|Rest])),
                Current is New + Old,
                    assert(offset_stack([Current,Old|Rest])), !.

/*
Pop the offset stack to shift back to the previous effective buffer start.
*/

pop_offset :-
                    offset_stack([_|Rest]),
                    not(empty(Rest)),
                    retract(offset_stack([_|Rest])),
                    assert(offset_stack(Rest)), !.
pop_offset :-
                    writestring("cannot pop initial zero offset"), !, fail.

/*
Check to see if any of the attention shifting rules match.
*/

check_as_rules(BufferCell,Features) :-
                    peek((_,_,Packets,_)),
                    clause_or_np_level(Packets,BufferCell,Features), !.

clause_or_np_level(Packets,BufferCell,Features) :-
                    member(cpool,Packets),
                    as_rule(cpool,BufferCell,Features), !.
clause_or_np_level(Packets,BufferCell,Features) :-
                    member(npool,Packets),
                    as_rule(npool,BufferCell,Features), !.
clause_or_np_level(_,_,_) :- !.
```

```
/**************************************************************/
/* OPERATIONS ON PARSE NODES IN THE BUFFER OR STACK */
/**************************************************************/

/*
Add to a node's list of features. Since in most cases features are added to the
current active node, a call to 'label' with no argument will refer to this node by
default. Features may also be added to the dominating cyclic node or to nodes in
the buffer.
*/

label(NewFeatures) :-
                pop((NodeName,OldFeatures,Packets,Descendents)),
                append(NewFeatures,OldFeatures,CurrentFeatures),
                push((NodeName,CurrentFeatures,Packets,Descendents)), !.

label(BufferCell,NewFeatures) :-
                integer(BufferCell),
                cell_name(BufferCell,CellName),
                Cell =.. [CellName,(NodeName,OldFeatures,Descendents)],
                retract(Cell),
                append(NewFeatures,OldFeatures,CurrentFeatures),
                UpdatedCell =.. [CellName,(NodeName,CurrentFeatures,Descendents)],
                assert(UpdatedCell), !.

label(DCN,NewFeatures) :-
                retrieve_dcn(DCN,(NodeName,OldFeatures,Packets,Descendents),Pos),
                append(NewFeatures,OldFeatures,CurrentFeatures),
                retract(active_node_stack((Pos,_))),
                assert(active_node_stack((Pos,
                  (NodeName,CurrentFeatures,Packets,Descendents)))), !.

/*
Check to see if the specified parse node has the given feauture. The node may be
the current active node (can), the dominating cyclic node (s or np), or an element
of the buffer. Note that every time the parser checks an element of the buffer, it
first checks to see if that element triggers any attention shifting rule.
*/

has_feature(can,Feature) :-
                peek((_,Features,_,_)), !,
                member(Feature,Features), !.

has_feature(BufferCell,Feature) :-
                integer(BufferCell),
                read(BufferCell,(_,Features,_)),
                check_as_rules(BufferCell,Features),
                read(BufferCell,(_,PossiblyChangedFeatures,_)), !,
                member(Feature,PossiblyChangedFeatures), !.

has_feature(DCN,Feature) :-
                retrieve_dcn(DCN,(_,Features,_,_),_), !,
                member(Feature,Features), !.
```

```
/*
Assign to the current active node whichever of the given possible features the first
element of the buffer has.
*/

transfer(PossibleFeatures) :-
            read(1,(_,Features,_)),
            intersection(Features,PossibleFeatures,CanFeatures),
            label(CanFeatures), !.
```

```
/**********************/
/* INPUT ROUTINES */
/**********************/

/*
The user is prompted to type in a sentence to be parsed. Readline gets every
character up to a carriage routine and leaves them as a list of characters in its
argument. Readline1 stops the recursion. The list will be passed to the routines
comprising the lexical analyzer to be transformed into a list of PROLOG atoms
representing each word. This list is asserted into the database for access by the
parser. Upon completion, what remains of the list (if anything) is retracted and the
user is asked whether he wishes to continue.
*/

input :-
                !,
                clear,
                nl,write('Sentence to parse'),
                nl,write(' > '),
                readline(Chars),
                sentence(Chars), !.

readline(Chars) :-
                get0(Ch),
                readline1(Ch,Chars).

readline1(10,[]) :- !.
readline1(Ch,[Ch|Chars]) :-
                readline(Chars), !.

sentence(Chars) :-
                tokens(Atoms,Chars,[]), !,
                assert(input_list(Atoms)),
                parse,
                retract(input_list(_)), !,
                nl,nl,
                write('Carry on? y/n : '),
                get0(X),get0(10),
                name(Ans,[X]),
                again(Ans), !.

again(y) :- input, !.
again(_) :- !.

parse :-
                initial_rule(Tree),
                print_tree(Tree).
```

```
/*************************/
/* LEXICAL ANALYZER  */
/*************************/

/*
The following definite clause grammar provides the scanning and tokenizing of a
sentence input as a list of characters and passes words and punctuation back as a
list of atoms.
*/

tokens(Atoms)          --> space, !, tokens(Atoms).
tokens([Atom|Atoms]) --> token(Atom), !, tokens(Atoms).
tokens([])           --> [].

token(Atom)          --> word(Chars), !, { name(Atom,Chars) }.
token(Integer)       --> constant(Integer), !.
token(Punct)         --> punctuation(Punct), !.

space                --> " ".
space                --> [10]. /* carriage return */

num(N)                 --> number(Number), !, { name(N,Number) }.

number([D|Ds])         --> digit(D), digits(Ds).

digit(D)             --> [D], { is_digit(D) }.

is_digit(D)          :- D>47, D<58. /* 0-9 */

digits([D|Ds])       --> digit(D), digits(Ds).
digits([])           --> [].

word([L|Ls])          --> letter(L), lords(Ls).

letter(L)            --> [L1], { is_letter(L1,L) }.

is_letter(L,L)       :- L>96, L<123, !. /* a-z */
is_letter(L1,L)      :- upper_case(L1), L is L1+32, !.

upper_case(L)         :- L>64, L<91. /* A-Z */

lords([L|Ls])        --> ( letter(L) ), lords(Ls).
lords([L|Ls])        --> ( digit(L) ), lords(Ls).
lords([])            --> [].

constant(C)           --> num(C), !.

punctuation('.')    --> "." , !.
punctuation('?')    --> "?" , !.
punctuation('''')   --> "'", !.
punctuation(',')    --> "," , !.
punctuation('!')    --> "!" , !.
```

```
/************************/
/* UTILITY ROUTINES */
/************************/

append([],L,L) :- !.
append([X|R],L,[X|R1]) :- append(R,L,R1).

/*
Clear the database before a new parse.
*/

clear :-
            make_empty_buffer,
            zero_buffer_offset,
            make_empty_stack,
            retract_all(currnum(_,_)),
            retract_all(input_list(_)).

/*
Create a new unique constituent name by concatenating the given type with a
unique number.
*/

conname(Type,Name) :-
            get_num(Type,Num),
            name(Type,Typechars),
            integer_name(Num,Numchars),
            append(Typechars,Numchars,Namechars),
            name(Name,Namechars).

/*
Delete every occurrence of the first argument from the second (a list).
*/

delete(_,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).

delete_all([],L,L).
delete_all([H|T],L,M) :-
            delete(H,L,L1),
            delete_all(T,L1,M).

empty([]).

empty_node(([],[],[])).

/*
Generate a unique number.
*/

get_num(Type,Num) :-
            retract(currnum(Type,Num1)), !,
            Num is Num1+1,
                asserta(currnum(Type,Num)).
get_num(Type,1) :-
                asserta(currnum(Type,1)).

/*
Convert an integer to a list of characters
*/

integer_name(I,List) :-
                integer_name(I,[],List).
integer_name(I,SoFar,[C|SoFar]) :-
                I<10, !, C is I+48.
integer_name(I,SoFar,List) :-
                Top is I//10,
                Bot is I mod 10,
                C is Bot+48,
                integer_name(Top,[C|SoFar],List).

/*
Find the intersection of two sets represented as lists.
*/

intersection([],X,[]).
intersection([X|R],Y,[X|Z]) :-
                member(X,Y), !,
                intersection(R,Y,Z).
intersection([X|R],Y,Z) :-
                intersection(R,Y,Z).

lookup(Word,Features) :-
                lex(Word,Features), !.

member(X,[X|_]) :- !.
member(X,[_|Y]) :- member(X,Y).

/*
Print out the final parse tree.
*/

print_tree(Tree) :-
                nl,
                pretty_print(Tree,0).

pretty_print((_,Features,_,Descendents),I) :- !,
                spaces(I),
                write(s),
                write(': '),
                print_features(Features),
                I4 is I + 4,
                print_descendents(Descendents,I4).

print_descendents([],_) :- !.
print_descendents([(Node,Features,Descendents)|Rest],I) :-
                nl,
                spaces(I),
                write(Node),
                write(:),
                print_words_or_own_descendents(Features,Descendents,I),
                print_descendents(Rest,I), !.

print_features([]) :- !.
print_features(Features) :-
                spaces(1),
                write(Features), !.

print_words_or_own_descendents(Features,Word,_) :-
                atom(Word),
                nil_word(Word,Features), !.
print_words_or_own_descendents(Features,[Word|Words],_) :-
                atom(Word),
                print_words([Word|Words]), !.
print_words_or_own_descendents(Features,Descendents,I) :-
                print_features(Features),
                I4 is I + 4,
                print_descendents(Descendents,I4), !.

print_words([]) :- !.
print_words([Word|Words]) :-
                spaces(1),
                write(Word),
                print_words(Words), !.

nil_word([],Features) :-
                print_features(Features), !.

nil_word(Word,_) :-
                spaces(1),
                write(Word), !.

spaces(0) :- !.
spaces(N) :-
    write(' '),
    N1 is N - 1,
    spaces(N1).

retract_all(X) :-
                retract(X), fail.
retract_all(_) :- !.

writestring([]).
writestring([N|Ns]) :-
                name(Name,[N]),
                write(Name),
                writestring(Ns).
```

```
/*************/
/* LEXICON */
/*************/

/*
The general form of an entry is:

            lex(<word>,<features>)

<features> is a list containing the root of the word, its person/number, its part of
speech, its tense, the types of objects and complements it takes, and any other
necessary information. A list of the possible features is given elsewhere.
*/

/* ADJECTIVES */

lex(happy,['*happy',adj]).

/* DETERMINERS */

lex(a,['*a',ns,n3p,det,indef,ngstart]).
lex(an,['*a',ns,n3p,det,indef,ngstart]).
lex(the,['*the',ns,npl,n3p,det,def,ngstart]).


/* NOUNS */

lex(book,['*book',ns,noun,ngstart]).
lex(cover,['*cover',ns,noun,ngstart]).
lex(exam,['*exam',ns,noun,ngstart]).
lex(executives,['*executive',npl,noun,ngstart]).
lex(lecture,['*lecture',ns,noun,ngstart]).
lex(meeting,['*meeting',ns,noun,ngstart]).
lex(tomorrow,['*tomorrow',ns,n3p,noun,ngstart,time]).
lex(yesterday,['*yesterday',ns,n3p,noun,ngstart,time]).

/* PREPOSITIONS */

lex(before,['*before',prep]).
lex(by,['*by',prep]).
lex(for,['*for',prep,comp]).
lex(from,['*from',prep]).
lex(in,['*in',prep]).
lex(of,['*of',prep]).
lex(on,['*on',prep]).
lex(to,['*to',prep,auxverb]).
lex(with,['*with',prep]).

/* PRONOUNS */

lex(i,['*I',ns,n1p,noun,pronoun,ngstart]).
lex(you,['*you',ns,npl,n2p,noun,pronoun,ngstart]).
lex(he,['*he',ns,n3p,noun,pronoun,ngstart]).
lex(she,['*she',ns,n3p,noun,pronoun,ngstart]).
lex(it,['*it',ns,n3p,noun,pronoun,ngstart]).
lex(we,['*we',npl,n1p,noun,pronoun,ngstart]).
lex(they,['*they',npl,n3p,pronoun,ngstart]).

lex(that,['*that',ns,npl,n3p,pronoun,relpron,comp]).
lex(what,['*what',ns,npl,n3p,np,pronoun,det,wh]).
lex(when,['*when',ns,np,pronoun,wh,time]).
lex(who,['*who',ns,n3p,np,pronoun,relpron,wh]).

/* PROPER NOUNS */

lex(john,['*john',ns,n3p,name,noun,propnoun,ngstart]).
lex(mary,['*mary',ns,n3p,name,noun,propnoun,ngstart]).
lex(smith,['*smith',ns,n3p,name,noun,propnoun,ngstart]).
lex(vancouver,['*vancouver',ns,n3p,noun,place,ngstart]).
lex(wednesday,['*wednesday',ns,n3p,noun,dow,ngstart]).

/* PUNCTUATION */

lex(',',['*,',punc]).
lex('''',['*''',punc]).
lex('!',['*!',finalpunc]).
lex('.',['*.',finalpunc]).
lex('?',['*?',finalpunc]).

/* TITLES */

lex(mr,['*mr',title]).
lex(mrs,['*mrs',title]).

/* VERBS */

lex(be,['*be',vspl,verb,auxverb,tnsless]).
lex(am,['*be',v1s,verb,auxverb,pres]).
lex(are,['*be','vpl+2s',verb,auxverb,pres]).
lex(is,['*be',v3s,verb,auxverb,pres]).
lex(was,['*be','v+13s',verb,auxverb,past]).
lex(were,['*be','vpl+2s',verb,auxverb,past]).
lex(been,['*be',vspl,verb,past,en]).
lex(being,['*be',vspl,verb,pres,part,ing]).

lex(do,['*do','v-3s',verb,auxverb,pres,tnsless]).
lex(does,['*do',v3s,verb,auxverb,pres]).
lex(did,['*do',vspl,verb,auxverb,past]).
lex(doing,['*do',vspl,verb,auxverb,pres,ing]).
lex(done,['*do',vspl,verb,auxverb,past,part,en]).

lex(give,['*give','v-3s',verb,pres,tnsless,'inf-obj','to-less-inf-obj']).
lex(gives,['*give',v3s,verb,pres,'inf-obj','to-less-inf-obj']).
lex(gave,['*give',vspl,verb,past,'inf-obj','to-less-inf-obj']).
lex(giving,['*give',vspl,verb,pres,part,ing,'inf-obj','to-less-inf-obj']).
lex(given,['*give',vspl,verb,past,part,en,'inf-obj','to-less-inf-obj']).

lex(have,['*have','v-3s',verb,auxverb,pres,tnsless]).
lex(has,['*have',v3s,verb,auxverb,pres]).
lex(had,['*have',vspl,verb,auxverb,past,en]).
lex(having,['*have',vspl,verb,auxverb,pres,part,ing]).

lex(help,['*help','v-3s',verb,pres,tnsless,'inf-obj','to-less-inf-obj',
        'subj-less-inf-obj']).
lex(helps,['*help',v3s,verb,pres,'inf-obj','to-less-inf-obj',
        'subj-less-inf-obj']).
lex(helped,['*help',vspl,verb,past,part,en,'inf-obj','to-less-inf-obj',
        'subj-less-inf-obj']).
lex(helping,['*help',vspl,verb,pres,part,ing,'inf-obj','to-less-inf-obj',
        'subj-less-inf-obj']).

lex(hit,['*hit','v-3s',verb,pres,tnsless,'comp-obj']).
lex(hits,['*hit',v3s,verb,pres,'comp-obj']).
lex(hit,['*hit',vspl,verb,past,part,en,'comp-obj']).
lex(hitting,['*hit',vspl,verb,pres,part,ing,'comp-obj']).

lex(persuade,['*persuade','v-3s',verb,pres,tnsless,'inf-obj']).
lex(persuades,['*persuade',v3s,verb,pres,'inf-obj']).
lex(persuaded,['*persuade',vspl,verb,past,part,en,'inf-obj']).
lex(persuading,['*persuade',vspl,verb,pres,part,ing,'inf-obj']).

lex(say,['*say','v-3s',verb,pres,tnsless,'comp-obj']).
lex(says,['*say',v3s,verb,pres,'comp-obj']).
lex(said,['*say',vspl,verb,past,en,'comp-obj']).
lex(saying,['*say',vslp,verb,pres,part,ing,'comp-obj']).

lex(see,['*see','v-3s',verb,pres,tnsless,'comp-obj']).
lex(sees,['*see',v3s,verb,pres,'comp-obj']).
lex(saw,['*see',vspl,verb,past,'comp-obj']).
lex(seeing,['*see',vspl,verb,pres,part,ing,'comp-obj']).
lex(seen,['*see',vspl,verb,past,en,'comp-obj']).

lex(seems,
  ['*seem',v3s,verb,pres,'no-subj','that-obj','inf-obj','to-be-less-inf-obj']).

lex(schedule,['*schedule','v-3s',verb,pres,tnsless,'comp-obj','inf-obj']).
lex(schedules,['*schedule',v3s,verb,pres,'comp-obj','inf-obj']).
lex(scheduled,['*schedule',vspl,verb,past,part,en,'comp-obj','inf-obj']).
lex(scheduling,['*schedule',vspl,verb,pres,part,ing,'comp-obj','inf-obj']).

lex(should,['*should',vspl,verb,auxverb,past,modal]).

lex(take,['*take','v-3s',verb,pres,tnsless,'inf-obj']).
lex(takes,['*take',v3s,verb,pres,'inf-obj']).
lex(took,['*take',vspl,verb,past,'inf-obj']).
lex(taking,['*take',vspl,verb,pres,part,ing,'inf-obj']).
lex(taken,['*take',vspl,verb,past,part,en,'inf-obj']).

lex(tell,['*tell','v-3s',verb,pres,tnsless,'inf-obj']).
lex(tells,['*tell',v3s,verb,pres,'inf-obj']).
lex(told,['*tell',vspl,verb,past,part,en,'inf-obj']).
lex(telling,['*tell',vspl,verb,pres,part,ing,'inf-obj']).
lex(want,['*want','v-3s',verb,pres,tnsless,'inf-obj','subj-less-inf-obj']).
lex(wants,['*want',v3s,verb,pres,'inf-obj','subj-less-inf-obj']).
lex(wanted,['*want',vspl,verb,past,part,en,'inf-obj','subj-less-inf-obj']).
lex(wanting,['*want',vspl,verb,pres,part,ing,'inf-obj','subj-less-inf-obj']).

lex(will,['*will',vspl,verb,auxverb]).

lex(would,['*would',vspl,verb,auxverb,past,modal]).

lex(Word,_) :-
            write(Word),
            writestring(" is not in the lexicon"), nl, fail.
/**************/
/* FEATURES */
/**************/

/*
auxverb         % auxiliary verb
comp            % complement markers like "that" and "for"
comp-np         % NP node dominating a complement S
comp-obj        % verb takes a complement as object
comp-s          % S serving as a complement
copula          % copular auxiliary
decl            % declarative sentence
def             % definite article or NP
det             % determiner
dow             % day of the week
en              % verb with an "en" ending however spelled
finalpunc       % final punctuation mark
future          % future tense
imper           % imperative sentence
indef           % indefinite article or NP
inf             % infinitive verb form
inf-obj         % verb takes an infinitive complement as object
inf-s           % infinitive S
ing             % verb with "ing" ending however spelled
major           % major S
modal           % either a modal or aux with an attached modal; e.g. should
n1p             % 1st person noun
n2p             % 2nd person noun
n3p             % 3rd person noun
name            % a person's name; e.g., John Smith
nbar            % "N-bar" node
ngstart         % anything that could start a noun group
noun            % any noun
no-subj         % verbs w/ delta subjects (or "it"); e.g., seems
not-modifiable  % NP which cannot take restrictive modifiers
np              % noun phrase
np-quest        % question with a fronted NP
np-utterance    % utterance consisting only of an NP
npl             % plural noun or quantifier (e.g., some)
ns              % singular noun, determiner (e.g., a), etc.
part            % participle
passive         % passive verb
```

```
past               % past tense
perf               % perfective
pp                 % prepositional phrase
pp-quest           % question with a fronted PP
pp-utterance       % utterance consisting only of a PP
pred-verb          % anything which can introduce a predicate after a copula
prep               % preposition
pres               % present tense
prog               % progressive
pronoun            % pronoun
pron-np            % NP that dominates a pronoun; therefore, not modifiable
propnoun           % proper noun
propn-np           % NP that dominates a proper noun; therefore, not modifiable
relpron            % relative pronoun
relpron-np         % NP that dominates a relative noun; therefore not modifiable
quest              % any kind of question
sec                % secondary S - not major - embedded
subj-less-inf-obj % verbs like want
that-obj           % verb takes a tensed complement
that-s             % embedded finite complement
time               % time word or phrase
tnsless            % tenseless verb
to-be-less-inf-obj% verb takes an infinitive w/o "to be"; e.g., seems
to-less-inf-obj    % verb takes an infinitive w/o "to"; e.g., help
trace              % NP which is a trace
utilized           % indicates the gap corresponding to a whcomp has been found
v+13s              % verb agrees with a 1st or 3rd person singular noun
v-3s               % verb matches any noun except 3rd person singular
v1s                % verb agrees only with 1st person singular noun; e.g., am
v3s                % verb agrees only with 3rd person singular noun; e.g., is
vpl+2s             % verb agrees w/ any plural or 2nd sing. noun; e.g., are
vspl               % agrees with any noun, singular or plural
verb               % any kind of verb
vp                 % verb phrase
wh                 % either a det with a wh marker or an NP with such a det
whcomp             % any sort of wh phrase
wh-quest           % wh question
yn-quest           % yes/no question
*/
```

Appendix 2
Sample Parses

The following examples show the types of sentences this **Prolog** implementaion of **PARSIFAL** is currently able to parse. They include the linguistic generalizations discussed in chapter four which in some instances, involve the use of traces.

```
/* Simple declarative sentence with prepositional phrase modifier */
 > John Smith has scheduled the meeting for Wednesday.

s: [decl,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
      noun: smith
   aux: [perf,v3s,pres,aux]
      perf: has
   vp: [vp]
      verb: scheduled
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   pp: [time,pp]
      prep: for
      np: [time,dow,np]
         noun: wednesday
   finalpunc: .

/* WH question : subject */
 > Who scheduled the meeting?

s: [utilized,np-quest,quest,wh-quest,major,s]
   whcomp: who
   np: [trace,not-modifiable,np]
      bound to: who
   aux: [vspl,past,aux]
   vp: [vp]
      verb: scheduled
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   Finalpunc: ?
```

```
/* NP utterance */
 > John.

s: [np-utterance,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   finalpunc: .

/* WH question : object */
 > What did John schedule?

s: [utilized,np-quest,quest,wh-quest,major,s]
   whcomp: what
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [vspl,past,aux]
      do: did
   vp: [vp]
      verb: schedule
      np: [trace,not-modifiable,np]
         bound to: what
   finalpunc: ?

/* NP utterance */
 > A meeting.

s: [np-utterance,s]
   np: [indef,det,np]
      det: a
      nbar: [ns,nbar]
         noun: meeting
   finalpunc: .

/* WH question : preposition */
 > When did he schedule it for?

s: [utilized,pp-quest,quest,wh-quest,major,s]
   whcomp: when
   np: [ns,n3p,pron-np,not-modifiable,np]
      pronoun: he
   aux: [vspl,past,aux]
      do: did
   vp: [vp]
      verb: schedule
      np: [ns,n3p,pron-np,not-modifiable,np]
         pronoun: it
      pp: [pp]
         prep: for
         np: [trace,np]
            bound to: when
   finalpunc: ?
```

```
/* PP utterance */
 > For Wednesday.

s: [pp-utterance,s]
   pp: [pp]
      prep: for
      np: [time,dow,np]
         noun: wednesday
   finalpunc: .

/* WH question */
 > What did he do?

s: [utilized,np-quest,quest,wh-quest,major,s]
   whcomp: what
   np: [ns,n3p,pron-np,not-modifiable,np]
      pronoun: he
   aux: [vspl,past,aux]
      do: did
   vp: [vp]
      verb: do
      np: [trace,not-modifiable,np]
         bound to: what
   finalpunc: ?

/* Verb cluster with no auxiliary */
 > John scheduled the meeting.
s: [decl,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [vspl,past,aux]
   vp: [vp]
      verb: scheduled
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   finalpunc: .

/* Modal, perfective auxiliary */
 > John should have scheduled the meeting.
s: [decl,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [perf,modal,vspl,past,aux]
      modal: should
      perf: have
   vp: [vp]
      verb: scheduled
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   finalpunc: .
```

```
/* Yes/No question with inversion of future auxiliary */
  > Will Mary give a lecture in Vancouver?

s: [quest,yn-quest,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: mary
   aux: [future,vspl,aux]
      will: will
   vp: [vp]
      verb: give
      np: [indef,det,np]
         det: a
         nbar: [ns,nbar]
            noun: lecture
   pp: [place,pp]
      prep: in
      np: [place,np]
         nbar: [ns,n3p,nbar]
            noun: vancouver
   finalpunc: ?

/* Yes/No question with inversion of perfective auxiliary */
  > Has Mr. Smith scheduled the meeting?

s: [quest,yn-quest,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      title: mr
      noun: smith
   aux: [perf,v3s,pres,aux]
      perf: has
   vp: [vp]
      verb: scheduled
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   finalpunc: ?

/* Yes/No question with inversion of 'do-support' auxiliary */
  > Did Mrs. Smith schedule the meeting?

s: [quest,yn-quest,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      title: mrs
      noun: smith
   aux: [vspl,past,aux]
      do: did
   vp: [vp]
      verb: schedule
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: meeting
   finalpunc: ?
```

```
/* Passive (trace feature) */
 > The meeting has been scheduled.

s: [np-preposed,decl,major,s]
   np: [def,det,np]
      det: the
      nbar: [ns,nbar]
         noun: meeting
   aux: [passive,perf,v3s,pres,aux]
      perf: has
      passive: been
   vp: [vp]
      verb: scheduled
      np: [trace,not-modifiable,np]
         bound to: the meeting
   finalpunc: .

/* Y/N quest w/ subj separating 2 parts of a progressive aux */
 > Is John scheduling a meeting for tomorrow?

s: [quest,yn-quest,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [prog,v3s,pres,aux]
      prog: is
   vp: [vp]
      verb: scheduling
      np: [indef,det,np]
         det: a
         nbar: [ns,nbar]
            noun: meeting
   pp: [time,pp]
      prep: for
      np: [time,np]
         nbar: [ns,n3p,time,nbar]
            noun: tomorrow
   finalpunc: ?

/* Y/N quest w/ passive verb and subj separating 2 parts of progressive aux */
 > Is a meeting being scheduled?

s: [np-preposed,quest,yn-quest,major,s]
   np: [indef,det,np]
      det: a
      nbar: [ns,nbar]
         noun: meeting
   aux: [passive,prog,v3s,pres,aux]
      prog: is
      passive: being
   vp: [vp]
      verb: scheduled
      np: [trace,not-modifiable,np]
         bound to: a meeting
   finalpunc: ?
```

```
/* Imperative ("you" insertion) */
 > Schedule a meeting for Wednesday!

s: [imper,major,s]
   np: [ns,npl,n2p,pron-np,not-modifiable,np]
      pronoun: you
   aux: [v-3s,pres,tnsless,aux]
   vp: [vp]
      verb: schedule
      np: [indef,det,np]
         det: a
         nbar: [ns,nbar]
            noun: meeting
   pp: [time,pp]
      prep: for
      np: [time,dow,np]
         noun: wednesday
   finalpunc: !

/* Simple embedded complement */
 > We wanted John to schedule the meeting.

s: [decl,major,s]
   np: [npl,n1p,pron-np,not-modifiable,np]
      pronoun: we
   aux: [vspl,past,aux]
   vp: [vp]
      verb: wanted
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [name,ns,n3p,not-modifiable,np]
               noun: john
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: schedule
               np: [def,det,np]
                  det: the
                  nbar: [ns,nbar]
                     noun: meeting
   finalpunc: .
```

```
/* Passive construction with embedded complement */
  > The meeting of the executives seems to have been scheduled for Wednesday.

s:  [np-preposed,decl,major,s]
    np: [def,det,np]
        det: the
        nbar: [ns,nbar]
            noun: meeting
            pp: [pp]
                prep: of
                np: [def,det,np]
                    det: the
                    nbar: [npl,nbar]
                        noun: executives
    aux: [v3s,pres,aux]
    vp: [vp]
        verb: seems
        np: [comp-np,np]
            s: [np-preposed,sec,comp-s,inf-s,s]
                np: [trace,not-modifiable,np]
                    bound to: the meeting of the executives
                aux: [passive,perf,inf,aux]
                    to: to
                    perf: have
                    passive: been
                vp: [vp]
                    verb: scheduled
                    np: [trace,not-modifiable,np]
                        bound to: the meeting of the executives
                pp: [time,pp]
                    prep: for
                    np: [time,dow,np]
                        noun: wednesday
    finalpunc: .

  > Who did John see?

s:  [utilized,np-quest,quest,wh-quest,major,s]
    whcomp: who
    np: [name,ns,n3p,not-modifiable,np]
        noun: john
    aux: [vspl,past,aux]
        do: did
    vp: [vp]
        verb: see
        np: [trace,not-modifiable,np]
            bound to: who
    finalpunc: ?
  > Who saw Mary?

s: [utilized,np-quest,quest,wh-quest,major,s]
   whcomp: who
   np: [trace,not-modifiable,np]
      bound to: who
   aux: [vspl,past,aux]
   vp: [vp]
      verb: saw
      np: [name,ns,n3p,not-modifiable,np]
         noun: mary
   finalpunc: ?

/* Verb taking an infinitive complement without an explicit "to" */
  > I gave Mary a book.

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [vspl,past,aux]
   vp: [vp]
      verb: gave
      np: [name,ns,n3p,not-modifiable,np]
         noun: mary
      np: [indef,det,np]
         det: a
         nbar: [ns,nbar]
            noun: book
   finalpunc: .

  > Who did John give the book to?

s: [utilized,np-quest,quest,wh-quest,major,s]
   whcomp: who
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [vspl,past,aux]
      do: did
   vp: [vp]
      verb: give
      np: [def,det,np]
         det: the
         nbar: [ns,nbar]
            noun: book
      pp: [pp]
         prep: to
         np: [trace,np]
            bound to: who
   finalpunc: ?
```

```
/* Verb taking a complement with an implicit "to"; "to" inserted */
  > What did John give Mary?
s:  [utilized,np-quest,quest,wh-quest,major,s]
    whcomp: what
    np: [name,ns,n3p,not-modifiable,np]
        noun: john
    aux: [vspl,past,aux]
        do: did
    vp: [vp]
        verb: give
        np: [trace,not-modifiable,np]
            bound to: what
        pp: [pp]
            prep: to
            np: [name,ns,n3p,not-modifiable,np]
                noun: mary
    finalpunc: ?

  > I saw the cover of the book.
s:  [decl,major,s]
    np: [ns,n1p,pron-np,not-modifiable,np]
        pronoun: i
    aux: [vspl,past,aux]
    vp: [vp]
        verb: saw
        np: [def,det,np]
            det: the
            nbar: [ns,nbar]
                noun: cover
                pp: [pp]
                    prep: of
                    np: [def,det,np]
                        det: the
                        nbar: [ns,nbar]
                            noun: book
    finalpunc: .

  > I hit Mary with a happy book.
s:  [decl,major,s]
    np: [ns,n1p,pron-np,not-modifiable,np]
        pronoun: i
    aux: [v-3s,pres,tnsless,aux]
    vp: [vp]
        verb: hit
        np: [name,ns,n3p,not-modifiable,np]
            noun: mary
        pp: [pp]
            prep: with
            np: [indef,det,np]
                det: a
                 adj: happy
                nbar: [ns,nbar]
                    noun: book
    finalpunc: .
```

```
/* Verb taking an infinitive object without "to be" */
 > John seems happy.

s: [np-preposed,decl,major,s]
   np: [name,ns,n3p,not-modifiable,np]
      noun: john
   aux: [v3s,pres,aux]
   vp: [vp]
      verb: seems
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [trace,not-modifiable,np]
               bound to: john
            aux: [copula,inf,aux]
               to: to
               copula: be
            vp: [vp]
               verb: happy
   finalpunc: .

 > Schedule John to give a lecture on Wednesday.

s: [imper,major,s]
   np: [ns,npl,n2p,pron-np,not-modifiable,np]
      pronoun: you
   aux: [v-3s,pres,tnsless,aux]
   vp: [vp]
      verb: schedule
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [name,ns,n3p,not-modifiable,np]
               noun: john
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: give
               np: [indef,det,np]
                  det: a
                  nbar: [ns,nbar]
                     noun: lecture
            pp: [time,pp]
               prep: on
               np: [time,dow,np]
                  noun: wednesday
   finalpunc: .
```

```
/* That complement */
 > It seems that a meeting has been scheduled.

s: [decl,major,s]
   np: [ns,n3p,pron-np,not-modifiable,np]
      pronoun: it
   aux: [v3s,pres,aux]
   vp: [vp]
      verb: seems
      np: [comp-np,np]
         s: [np-preposed,comp-s,that-s,sec,s]
            comp: that
            np: [indef,det,np]
               det: a
               nbar: [ns,nbar]
                  noun: meeting
            aux: [passive,perf,v3s,pres,aux]
               perf: has
               passive: been
            vp: [vp]
               verb: scheduled
               np: [trace,not-modifiable,np]
                  bound to: a meeting
   finalpunc: .

/* That complement without an explicit "that" */
 > It seems a meeting has been scheduled.

s: [decl,major,s]
   np: [ns,n3p,pron-np,not-modifiable,np]
      pronoun: it
   aux: [v3s,pres,aux]
   vp: [vp]
      verb: seems
      np: [comp-np,np]
         s: [np-preposed,comp-s,that-s,sec,s]
            np: [indef,det,np]
               det: a
               nbar: [ns,nbar]
                  noun: meeting
            aux: [passive,perf,v3s,pres,aux]
               perf: has
               passive: been
            vp: [vp]
               verb: scheduled
               np: [trace,not-modifiable,np]
                  bound to: a meeting
   finalpunc: .
```

```
/* Verb taking an infinitive embedded complement */
 > I helped John to do it.

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [vspl,past,aux]
   vp: [vp]
      verb: helped
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [name,ns,n3p,not-modifiable,np]
               noun: john
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: do
               np: [ns,n3p,pron-np,not-modifiable,np]
                  pronoun: it
   finalpunc: .

/* Verb taking an infinitive embedded complement without "to"; "to" inserted */
 > I helped John do it.

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [vspl,past,aux]
   vp: [vp]
      verb: helped
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [name,ns,n3p,not-modifiable,np]
               noun: john
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: do
               np: [ns,n3p,pron-np,not-modifiable,np]
                  pronoun: it
   finalpunc: .
```

```
/* Verb taking an infinitive embedded complement without a subject */
 > I helped to do it.

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [vspl,past,aux]
   vp: [vp]
      verb: helped
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [trace,not-modifiable,np]
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: do
               np: [ns,n3p,pron-np,not-modifiable,np]
                  pronoun: it
   finalpunc: .

/* Verb taking an infinitive embedded complement without "to" or a subject */
 > I helped do it.

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [vspl,past,aux]
   vp: [vp]
      verb: helped
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [trace,not-modifiable,np]
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: do
               np: [ns,n3p,pron-np,not-modifiable,np]
                  pronoun: it
   finalpunc: .
```

```
/* Verb taking an infinitive embedded complement without an explicit subject */
 > I want to do it!

s: [decl,major,s]
   np: [ns,n1p,pron-np,not-modifiable,np]
      pronoun: i
   aux: [v-3s,pres,tnsless,aux]
   vp: [vp]
      verb: want
      np: [comp-np,np]
         s: [sec,comp-s,inf-s,s]
            np: [trace,not-modifiable,np]
            aux: [inf,aux]
               to: to
            vp: [vp]
               verb: do
               np: [ns,n3p,pron-np,not-modifiable,np]
                  pronoun: it
   finalpunc: !
```