1987

UGURU: a natural language UNIX consultant

John Hanson

Rochester Institute of Technology
School of Computer Science and Technology

UGURU: A Natural Language Unix Consultant

by John Douglas Hanson

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

John A. Biles 2/19/88
Professor John A. Biles

Peter G. Anderson 22 Feb 88
Professor Peter G. Anderson

Hugh K. Donaghy 2/22/88
Professor Hugh K. Donaghy

February 22, 1987

Title of Thesis *UGURU: A Natural Language Unix Consultant*

I, John Douglas Hanson, hereby grant permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

John Douglas Hanson
Feb. 24, 1988

ABSTRACT

UGURU is a natural language conversation program, implemented in Prolog, which can manage a wide knowledge base of facts about Unix. The range and wording of questions that it understands are based on surveys taken of students, mostly Unix beginners. UGURU is also designed to accept statements in English that can be added as facts to the knowledge base. Each fact is represented as a "binding set:" a verb-oriented semantic net with the characteristics of directed acyclic graphs. The main actions taken by UGURU are divided between two primary modules, a parser and a retriever. To produce a binding set from an input, the parser incorporates a new kind of object-oriented grammar of several levels, parallel tracing of distinct parse trees by independent units called recognizers, the concurrent use of both syntactic and semantic knowledge, and a "pragmatic criterion" that requires the system to mimic the sequence of human parsing. The retriever, invoked to answer input questions, seeks to match the binding set representing the question to a fact in the knowledge base by performing semantic transformations on the two sets.

KEYWORDS

natural language processing, natural language interface, artificial intelligence, knowledge representation, knowledge acquisition, question-answering systems

COMPUTING REVIEWS SUBJECT CATEGORIES

I.2.7. Natural Language Processing
I.2.4. Knowledge Representation Formalisms and Methods
I.2.1. Applications and Expert Systems

TABLE OF CONTENTS

I. Introduction 1

II. Background 5

III. Early Versions of UGURU 25

IV. The Parser: Conceptual Development 50

V. The Parser: Implementation 78

VI. The Retriever 107

VII. Top-Level View of the Components:
A Complete Trace of an Input 139

VIII. Results and Conclusions 150

Appendix 167
Bibliography 172

CHAPTER I

INTRODUCTION

This thesis describes the design and implementation of UGURU, a natural language query system that answers questions about Unix[1] commands. The range of information planned for its knowledge base is derived in large part from examination of surveys conducted of students at the Rochester Institute of Technology. On the surveys, the students, mostly freshman but also including some upperclassman and older beginners with Unix, provided questions that had arisen at some time or other during their work sessions in the Unix environment. Since the questions gathered in the surveys were to be formulated as questions ending with '?', they were also a basis for making judgments about the variety and complexity of natural language that UGURU would encounter. Also, in responding to the range of information covered by the survey questions, the system's design includes the facility for a user to add pieces of information to a user-accessible section of the knowledge base, either for personal recall later or sharing with other users.

[1] Unix is a trademark of AT&T Bell Laboratories.

UGURU is a knowledge-based system, rather than a domain-independent system. However, several versions of the program were written in the course of its development, and each new version has increased the emphasis on finding tools that have generality, and on studying what is necessary to produce that generality. Most of the important elements that make up this system are based on tools that are now standard in the field of natural language processing, though they are modified here with ideas that reflect recent trends in cognitive science and research into neural networks: in particular, the parallel aspects of thought processes and distributed representations of knowledge.

UGURU's operation is divided sequentially between two functions: the parser and the retriever. The parsing element relies on syntactic processing of input wherever it can, but it also incorporates concurrent semantic analysis to help disambiguate multiple parses, all of which are syntactically legal, or weak parses, which may occur when the sentence contains illegal or unorthodox syntax. The parser produces a connected graph - essentially a semantic net - in which related words in the input statement are linked to form the edges. The links most often represent the "deep cases" of the words involved. Superficially, these nets resemble trees, but since the edges are directed, always pointing to the modified word, many of the nets are more accurately DAGs (directed acyclic graphs). Each graph may comprise one or more trees that share members. Knowledge is represented by this same sort of semantic net. The retrieving function, which examines the knowledge base with the output of the parser as its input, uses transformational semantic procedures to identify equivalent semantic nets. It produces an answer to the original question by matching a net in the knowledge base to the net constructed by the parser.

One important function this system does not include is a mechanism for making inferences from statements, perhaps its greatest weakness. On the other hand, since the words of the dictionary are also represented in class hierarchies, there exists an implicit sort of syllogistic reasoning that exploits set membership or non-membership. Another way in which UGURU "reasons" about its choices is to use the constraints of the words themselves, based on common and acceptable usage of those words.

Design goals included a working system, of course, if possible. Some working principles of the design that would hopefully lead to that goal should be mentioned here. One is that the system's understanding of English would not be based on a set of primitive words (or symbols) into which all statements had to be translated. The point of view taken here is that such an arrangement is artificial, though it may be economical. Similarly, the decision was made to avoid the probable rigidity of frames or scripts, though they are among the few tools available to deal with contextual world knowledge. The burden of the work of understanding is, therefore, on procedures, and a possibly very large accumulation of procedures at that.

As a second working principle, the design of these functions exploits parallelism. PROLOG is the language chosen for the program, and, among its advantages in a project of this kind, PROLOG allows a handy implementation for simulating parallel activities. The representation of much of the knowledge base and semantic knowledge needed are also straight-forward in PROLOG.

In due consideration of the first two principles already mentioned, and also as as result of the experiences of earlier versions of UGURU, it became clear that the design had to have the capacity to allow additions to the procedures and knowledge base that could be made easily and without colliding with already existing elements. Lastly, the design chosen should allow the parser's understanding of a statement at any point to match as closely as possible the same understanding that a person would have on reading that statement up to that point. Actually satisfying this last priniciple would, of course, be nothing short of a miracle, but, as a way to guide the design of a system that expects to understand language, using the human mind as a model seems an appropriate choice. Given as flexible a framework as possible in which to add procedures and knowledge, attempting to work under this principle turns UGURU into a study of what is needed to achieve that understanding.

Chapter II discusses the background and sources on which this project has drawn. Chapter III covers the earlier versions of UGURU, which were discarded as their limitations became evident: these methods included keyword extraction, generative phrase structure

grammar, and a combination of a stack with ATNs (a simple and unintentional version of Marcus' "Wait and See Parser"). Chapter III also considers the set of survey questions, the conclusions drawn from them, and their effect on further development of the project.

The parsing function and its treatment of English are the topics of Chapters IV and V. Chapter IV, "Conceptual Development," discusses the approach taken to formulate a grammar for English. It offers an analysis of syntax on which the rules and procedures of the grammar are based. Chapter V discusses the implementation of the parser. Besides the mechanics, it also shows how the use of the grammar allows a fairly simple and consistent solution to several major problem areas in English NLP systems, including compound nouns, conjunctions, and ellipses. A brief description of the preprocessing of the input that precedes parsing will also be found there.

Chapter VI covers the retrieval function. It first describes the representation of knowledge and the knowledge base, and following that, the methods used to find answers to the questions. These methods are primarily the transformations performed on the output of the parser to locate matching expressions in the knowledge base of Unix facts. Finally, Chapter VII presents an overview of the top level of the program and traces an input through the system as a whole, and Chapter VIII presents results and conclusions.

CHAPTER II

BACKGROUND

The sources on which this project has drawn are varied. This is especially true since UGURU has passed through several stages, each exploiting different techniques in its implementation. Many of the ideas are based on standard tools and theories in the field of natural language processing, some of which may be considered conservative at this point in time. A number of ideas that developed in the course of the project turned out to have been explored or demonstrated already. The current version attempts to modify these standard theories to mimic the kind of parsing of natural language that the human brain performs, though this is far from completely understood. These modifications draw on new models of cognitive processing, and studies of the way we learn and use words. This chapter references these sources in the context of the issues of the design problem, and concludes with a brief description of UC, an existing program, which is a query system about Unix for beginners, and which relies on different techniques in its implementation from those used in UGURU.

2.1. Sources for Parsing Techniques

Ideas that have served as sources for parsing techniques in the several stages of UGURU's design come mainly from three standard models: the phrase structure grammar (PSG), augmented transition networks (ATNs), and the case grammar. Some additional ideas that played a part are perhaps a matter of common sense as much as theory, such as the use of constraints derived from the words themselves to aid parsing. These too have been elaborated formally, for example, the theory of preference semantics of Wilks [Wilk 78] or word expert parsing [Smal 80].

Eventually, most of these ideas were abandoned in favor of developing a parsing scheme in which syntactic and semantic processing could be integrated as much as possible.

2.1.1. Chomsky's Theories and the Phrase Structure Grammar

Ever since the theoretical work of Noam Chomsky [Chom 57], it has been generally agreed that a statement has an underlying "deep structure" of meaning with its own representational requirements. The manner of this representation was of less concern to Chomsky than the idea that a single deep structure could generate numerous surface structures - the actual words and phrasing used - whose relationship to the deep structure was governed by rules. The surface structures could be shown to be transformations of one another, also by means of rules. This systematic approach to understanding the structures of language, and his work on classifying sets of production rules, has had great influence on natural language processing.

At one stage of development, a parser was implemented for UGURU based on the type of grammar Chomsky identified as a "phrase structure grammar." It has been generally accepted for some time that natural language does not fall into the category of context free languages, so that this effort only could have succeeded if an acceptable subset of English

were used as input to the parser.[1] This version of UGURU also followed Chomsky's separation of syntactic processing and semantic processing. Hardly any of the semantic procedures were actually written, but all that were planned could only be executed upon completion of the parse tree. This is probably the aspect of Chomsky's work that has most often been called into question [Harr 85], [Katz 64], [Fodo 77].

While the current version of UGURU has departed from the notions that a complete set of phrase structure rules can be constructed to govern the parsing process, and that syntactic and semantic elements of language structure may be separately processed, it has taken another aspect of Chomsky's ideas to heart: that the problem of an underlying representation of meaning may be shifted, at least in part, to the problem of transformations that are allowed on it. Here the idea of transformation has been extended to semantics: different words or expressions that have the same meaning are considered transformations of one another, even if they are not rearrangements of the same stem words. The program treats the question of whether two expressions have the same meaning as a matter of heuristics, however, and no attempt has been made to try to systematize all transformations into rules.

The current design also uses another idea that Chomsky originated and that has become generally accepted. Clauses within a sentence are treated as embedded sentences, which have been transformed to appear as clauses in the surface structure. UGURU's knowledge is represented as semantic nets: each net may be a single clause, or a combination of clauses, but the representation of each clause is a meaningful entity that can stand alone.

2.1.2. ATNs

There were also partial implementations of UGURU using other major models of parsing in which syntactic analysis predominates. After the phrase structure grammar, the parsing function was constructed with ATNs, developed by Woods for the program LUNAR

[1] Gazdar [Gazd 83] and others argue that the question is still open and that phrase structure grammars have adequate power to drive a natural language parser. Chomsky himself, in the conclusions of *Syntactic Structures*, states that he does not believe that a PSG in itself can drive a natural language parser.

[Wood 70]. In his original design, the ATN grammar was a purely syntactic tool, but he has subsequently expanded the model to allow semantic checking to be integrated with the progression of state changes through the ATNs. The ATN framework itself does not handle the problems of the proper attachment of prepositional phrases and of conjunctions. Woods has suggested a mechanism to handle this which he calls "cascaded ATNs" [Wood 80].[2] These problems are too common to be avoided even in a very reduced natural language environment. Woods wrote that they must be handled through semantic processing, and that separate mechanisms must be created for this purpose. Marcus, designer of the wait-and-see parser, agrees with Woods on this point [Marc 79].

2.1.3. Some Difficulties of PSGs and ATNs

These two problems, the attachment of prepositional phrases and of conjunctions, and also the problem of the attachment of adjectival phrases, especially those beginning with gerunds, were among the main dissatisfactions with the ATN version of UGURU, as they had also been with the phrase structure grammar version.

Both versions also had approached parsing in a left-to-right, top-down manner. For the low-level syntactic patterns, such as noun phrases and prepositional phrases, parsing can proceed deterministically with a high degree of confidence. Both PSGs and ATN grammars are well suited for this. In recognizing elements of a higher level, however, top-down parsing errs much more often. Marcus gives the following example, though he is discussing the problems of nondeterministic parsers in general, and not just top-down parsers. He considers two sentences that begin with the word *have*.

Have the boys take the exam today.

Have the boys taken the exam today?

---

[2]There are, of course, numerous proposals for extending the basic ATN grammar without the addition of new constructs. Boguraev [Bogu 83] describes a model that parses conjunctions by giving the system the capability to dynamically create arcs on encountering a conjunction.

After encountering the word *have*, a nondeterministic parser must guess whether it should be taken as an imperative verb ([You] have the boys take the exam), as in the first case, or as an auxiliary beginning a question, as in the second. It is easy to expand this example, and most others like it. Without having seen any of the input sentence beyond that first word, a robust parser of English should be aware of other less frequent possibilities too from which it also must choose, such as:

Have the boys any last requests?

where *have* is the main verb of the question. The same issue surrounds the problem of attachment. The question that begins

How can I list the variable names in a file containing ...

may continue

numbers in them?

or

a Pascal program?

Where to attach *containing* is a decision that cannot be made securely until the rest of the clause has been seen. In the first case, it is the names that contain numbers, and in the second, the file that contains a Pascal program.

Marcus's goal was to develop a model of parsing that operated deterministically, or, in other words, that eliminated successfully all wrong guessing. Wrong guessing can be tolerated if either of two conditions exists to correct for it: backtracking occurs (suggesting a depth-first strategy for searching), or all alternative paths are expanded in parallel (suggesting a breadth-first strategy). Since parallelism historically has had to be simulated, Marcus refers to the second condition as pseudo-parallelism. In his view, a deterministic parser has no need for either backtracking or pseudo-parallelism, essentially by definition. Parsing without

the wrong guesses, except on garden path statements where people also backtrack, is desirable because it is psychologically realistic.

Other alternatives for overcoming the guessing problem include the use of lookahead, if the amount of lookahead can be determined and is not an excessively large amount of input: it has not been demonstrated that this can be done with natural language. There is the heuristic evidence of the Marcus model, however, that a small number of elements does in fact suffice for the parser to make decisions correctly.[3] Another alternative is changing to a bottom-up approach that allows unidentified elements to be saved and identification deferred until the decision is not in doubt. Bottom-up parsing can err also when there are ambiguous structures, and so it too requires a lookahead mechanism to operate deterministically. What is also possible is a combination of the top-down approach and the bottom-up approach. A version of UGURU that grew out of the ATN grammar implementation attempted to do just that. ATNs were used to pick out the low-level elements; these were placed in a stack for further reduction when that became possible.

2.1.4. The Wait-and-See Parser

Marcus' wait-and-see parser is an elegant formulation of this kind of an approach that combines bottom-up features, top-down features, and lookahead.[4] An implementation named PARSIFAL was announced by Marcus in 1980 [Marc 80]. Its grammar interpreter operates on two data structures, a stack and a three-cell buffer. The elements contained in either the stack or the buffer are nodes from the parse tree being constructed for the input. A node may be a single word (with lexical markers gathered from the lexicon), or a subtree representing some syntactic constituent, such as a noun phrase, subject, predicate or subordinate clause. Since the parser is deterministic, once a node is created, it is never unmade.

---

[3] An assumption about the input is necessary for this to hold true. The language of the input is language as used in speaking, and therefore generally less intricate than language that is written. Also, a mistaken parse is actually deemed correct on certain *garden path* sentences if that same sentence would mislead a person as well.

[4] Winston [Wins 84] considers the WASP parser to be entirely top-down with lookahead and demonstrably LR(k).

The business of the grammar interpreter is to identify nodes, and keep joining them until there is a single node left, the completed parse tree.

At any moment, the grammar interpreter is able to see the elements in the buffer (if any) and the top of the stack. This window of four elements constitutes the lookahead ability of the system. Preprocessing of a top-down nature identifies low-level constituents like noun phrases or prepositional phrases. A new element, whether a single word or low-level construct, enters the system through the buffer. There is a sense of flow between the stack and the buffer. Constituents are pushed onto the stack from the buffer, or attached to the top of the stack if their grammatical role is known and indicates that course of action. Nodes whose role is undetermined may be popped from the stack into the buffer.

The grammar rules that drive the interpreter are grouped into *rule packets*. Rule packets may be active or inactive, and the interpreter concerns itself only with rule packets that are active; as a constituent appears, it activates only the rule packets that potentially may be applied to it. Rules packets are attached to the constituents, so that if a constituent disappears by being submerged in the stack, its rule packets are also no longer visible, and thus become temporarily inactive.

There can be many advantages from designing a natural language processor so that system elements may be activated and deactivated. The idea shows itself in a number of forms in several existing systems: it may turn on the visibility or non-visibility of either procedural knowledge, like the rule packets in the wait-and-see parser, or of data structures. It may be implemented with pseudo-independent procedures, frequently called demons, that can self-destruct as they become irrelevant, as certain semantic processing is done in the *conceptual analyzers* developed with the conceptual dependency theory of Roger Schank [Scha 72], and that appear in programs like ELI and MARGIE [Scha 81]. In general, activating and deactivating system elements can increase efficiency. In the wait-and-see parser, the number of collisions of rules may also be reduced. Perhaps more importantly, with only a portion of the grammar rules active, the system takes on one of the characteristics associated with human

processing of language, the building of expectations: the system appears to be expecting certain kinds of behavior, and not others.

A great advantage of the system is the transparency of the rules, particularly in comparison with those of the phrase structure grammars or even ATNs. The rules can be written so that their effects can be visualized relatively easily. This is a very desirable feature, since many rules may contain ambiguities or overlap with other rules. In UGURU, the goal of transparent rules for parsing has been partially realized. While the logistics of the rules can be understood clearly, their coding is still at the same level (Prolog) as all other code in the system, and so knowledge of the various parameters involved is necessary to work with them.

While PARSIFAL has achieved a reasonable level of success and has been well received, there are still problems. It is a tool for providing syntactic analysis, and Marcus himself agrees that basic problems like the attachment of prepositional phrases and conjunctions require semantic processing and that mechanisms to provide this processing do not lie within the basic grammar structure. Also unsolved is the problem of lexical ambiguity, when one word can function as more than one part of speech. Marcus anticipates that extensions to the basic WASP grammar will have sufficient power to parse lexical ambiguities.

Others see some basic flaws in the design itself. Handling *garden path* sentences that people do without being fooled is a main objective of Marcus parsing, and a central motive for system's determinism. E. J. Briscoe [Bris 83] points out that looking for garden paths to solve by means of the lookahead contained in the buffer is actually a hard-wired feature of this kind of parsing, and that it finds garden paths where people don't because of their semantic knowledge. He concludes: "In fact, Marcus has offered a structural substitute for semantic pragmatic and prosodic information and thus distorted, rather than revealed, the nature of syntactic processing."

2.2. Source Models for Semantic Processing

Fillmore's case grammar [Fill 68] was one of the first theories to provide mechanisms for understanding semantic relationships within a comprehensive framework. The several versions of UGURU probably have more in common with this perspective than any of the others. A case grammar is still strongly oriented to using syntax to provide tools for dissecting language structures. Harris points out [Harr 85] that, beyond the structural relationships, Fillmore included

> notions about the *functional relationships* among the various phrases within a sentence, the part of syntax that conveys meaning.

The crucial issue in the theory concerns the changes in case that seem to result from rephrasing a sentence. The question arises naturally from comparing the transformations allowed by Chomsky's theory. The following sentences are an example.

> The hammer hit the nail.
>
> The nail was hit with the hammer.[5]

The surface cases of *nail* are different in the two sentences: objective in the first one, and nominative in the second. It should be clear that the actual relationship of *nail* and *hit* is the same in both sentences. Consequently understanding a sentence requires the capability to identify the "deep case" relationships that exist.

Fillmore presented a list of "deep cases" that included the agentive, instrumental, dative, factitive, locative, objective, and neutral. Other theorists have provided variations on the original list, including Fillmore himself. Fillmore also asserted that the main verb contains aspects of case which he called modality. This comprised information on tense, voice, and so forth. A sentence can then be considered a proposition plus its modality. The proposition

[5] Example from Harris [Harr 85], p. 37.

contains the verb and noun phrases that fill in the slots provided by the several cases. The set of cases is constant, and so can be regarded as a "case frame." The early designs for UGURU all envisioned filling the case frame as the goal of parsing. The current version has tried to take a more flexible view of the cases, with the opinion that a strict case frame is too rigid. Chapter IV will pursue this point further (Section 4.3.2.2).

There are two ideas of practical importance in Fillmore's scheme beside the notion of the "deep cases." The first is the notion that a proposition contains a verb plus attendant noun phrases governed by the cases: the verb is the focus with the nouns dependent on it. Secondly, while the cases are most often identified by an initial preposition, Fillmore theorized the existence of an abstract introductory symbol for each case, called the *Kasus*, that could take on the null value as well as the prepositions. The Kasus for the objective case is normally null, for example. While not starting out with these ideas, the parser design for UGURU exhibits the same viewpoint. In common also is the notion that at the highest level, identification of the verb and its dependents is crucial and the order of words is a function of a lower level. These ideas will be recognized in the discussions in Chapter IV.

The earliest descriptions of semantic nets [Quil 68] were published around the same time as Fillmore's pioneering work. An important feature they held in common with the case grammar was the notion that the verb is the structural focus of any thought. Simmons et al. [Simm 68] drew the connection between case grammars and semantic nets explicitly, and also added several important categories of relations. These included relations between non-verbs, such as superset, subset, equivalent, partof, etc. These ideas have been taken up subsequently by Rumelhart and Norman [Rume 73]. The current version of UGURU uses them as a central organizing principle. Their use provides a flexible data structure that fits well the needs of both the parsing and retrieval functions of the system.

2.2.1. The Model of Waltz and Pollack

This model differs from the approaches described so far on several key issues. Most importantly it integrates all processing, whether syntactic, semantic, or contextual. On this matter they state [Poll 86]:

> While these areas of knowledge seem distinct, it isn't easy to write a program for natural-language processing that decomposes language into its parts; i.e., you cannot construct a psychologically realistic natural-language processor by merely conjoining various knowledge-specific processing modules serially or hierarchically.

Judging from the inclusion of the phrase, "psychologically realistic," they apparently feel that a workable natural language processor should model human language processing.

To achieve the integration of the different sources of language knowledge, their model processes the various components in parallel.[6] All relevant information to the processing of an input is connected within a single network. The network grows with each additional word read from the input, by the inclusion of the word, lexical aspects of the word, syntactic aspects of the word, and related concepts and terms. The dynamic construction of the network is characterized by the phrase "spreading activation."

The network is a weighted graph where an edge indicates the degree of mutual activation or inhibition between the connected nodes. Both nodes and edges carry weights. Opposites, like *black* and *white*, are connected by a strongly inhibitory link that prevents both nodes from being active. A final "parse" of the sentence will therefore allow only those nodes to be active that are all mutually consistent. When enough information has been joined to the graph, the problems of ambiguity and garden paths will be resolved naturally, since the alternative and inconsistent parses will be deactivated. The weights on the nodes and edges are continually recalculated until the values stabilize at an overall consistency. The addition of all the nodes connected with a new input word begins a cycle of computations. During the cycle, a single node or edge weight may be recalculated many times.

[6] The name of one of their articles describing the model is *Massively Parallel Parsing*. [Walt85]

As an example of how the model works, the authors illustrate the sequence of steps taken to process the sentence, *The astronomer married a star.* The major problem in this sentence, which they call "a semantic garden path," is to identify which meaning of *star* should be activated. During the cycle of computations begun when the several meanings and associations of *star* are added to the graph, the weights initially favor the idea that the astronomer has married a celestial body. Gradually, the balance shifts to the idea that *star* means movie star. In this case, 85 recalculations are performed before the system reaches the correct conclusion.

The examples given in their articles are mostly concerned with the solution of "semantic garden paths," and the system is evidently quite successful in this area. They do not discuss the aspects of how syntactic analysis is performed, except to note that it is based on Kay's MIND system [Kay 73].

The implementation of contextual knowledge is another area where the Waltz and Pollack model differs significantly from the standard models. The issue is the way in which words are related to other words. The system obviously recognizes words, and the fact that many words have more than one meaning. When a word is activated, however, it must activate words that can be associated with it. The data base of contextual knowledge does not assert directly that two words are related. Instead, it contains two disjoint lists of words or phrases, one called "concepts" and the other "microfeatures," and the values of an incidence matrix that relates elements of one list to the other. Concepts are more general, microfeatures more specific. Nodes added to the activation network most often contain concepts. Microfeatures are chosen [Poll 86]

> on the basis of first principles to correspond to the major distinctions humans make about situations in the world ... For example, some important microfeatures correspond to distinctions such as threatening/safe, animate/inanimate, edible/inedible, indoors/outdoors ...

In particular, some microfeatures are second, minute, day, house, store, school, city street, forest, mountain, and seashore. Concepts include weekend, outdoors, fire at, waste money,

hunting, and dollar. The set of microfeatures is large, but established at several thousands. The list of concepts must grow as the system acquires more knowledge. The incidence matrix contains values that indicate the strength of the association between a given concept and microfeature. The value 1 indicates that the two are usually associated; 0.5 is a mild association; 0 indicates that the two might be associated, but generally are not. If the concept and microfeature are mutually exclusive the matrix contains the negative -0.5. The association values can be used in two ways. If two concepts are specified, their relative closeness can be computed directly by a mathematical function. Otherwise, if only a set of microfeatures is specified, a class of related concepts can be generated. In fact, the authors claim that hierarchies of classes of related concepts are automatically produced because of the nature of the implementation.

Another very important and timely aspect of this model is that it can be implemented on a parallel computer. Each node may be assigned an individual processor, with the computations performed through message passing. The authors have worked out the details of two designs for an appropriate hardware implementation, and indicate that they expect to install the system on the Connection Machine in the future.

2.3. Models of Human Language Processing

Even though it may be determined some time in the future that a computer is not suited to "think" in the same manner that a person thinks, the difficulties of natural language processing and the great amount of work yet to be done to find solutions suggest that it is a good idea to understand what we can about how people process language. This is also a subject about which much more is not known than is known, but there are studies and theories that could have applications to a project such as UGURU. The current version of UGURU has attempted to use many features of human language processing as a source for its design, and any subsequent improvements in the system would incorporate even more of these elements. Ideas concerning these features derived from the work of a variety of individuals, of whom many have a background in the computing field.

2.3.1. Manipulation and Learning of Words by People

In most NLP systems, the lexicon is accessed by a word. The word may have information of various sorts attached to it, structured in a variety of ways. The word may have multiple entries, when there are multiple meanings or grammatic functions. The word is the unit of finest granularity. There is reason to believe that this is not so with the brain.

Children learn words by context. For children in western cultures the context is of two kinds. First, a word is learned together with the words or ideas that surround it when it is encountered. When recalled, it is recalled along with the context. The word has no meaning, at least originally, apart from its associations. The other kind of context is the general class into which the word fits. In studies done by Miller and Gildea [Mill 87], children were presented with objects with unfamiliar colors, and told the name of the color only once. When asked later what the word meant, the children most frequently could identify that the word was a color, and the type of object from which they learned it, but not the particular hue. They suggest that adults have some of the same problems. Miller and Gildea made further tests in which children were given words along with the dictionary entries for those words, and then told to write sentences using the words. Although there was usually some logic to the answer, it was most frequently an incorrect use of the term. By finding a synonym in the dictionary entry, the children recalled the context for the synonym and automatically attached the new word to the old context, unaware of the conflicting connotations.

It is an important question whether or not using words implies using classes that contain the words. This kind of instinct may in fact be cultural. Studies by Cole [Rest 84] demonstrated that, while western children were helped to remember lists of arbitrary words by grouping them into classes, this technique was of no help at all when Liberian rice farmers were presented with the same problem. The Liberians were able to retain the list of words when they were incorporated into a story, however. To support the conceptual dependency theory, Schank has supported testing (on westerners, presumably) to show that people do

process classifications when they think [Schw 87]. The tests provided evidence that processing an idea about an object that is one-of-a-kind takes less time than processing an idea about an object that is very much part of a set of related objects. Piaget's theories on the stages of mental ability suggest that classifying objects is a learned technique, generally acquired about the age of seven. [Piag 52]

Two interesting studies have been done that aim to test the limits of thesauruses and dictionaries. Hardin [Dewd 87] produced chains of synonyms from thesauruses, which linked a word to its exact opposite. Every synonym in the link is a commonly accepted and natural substitute for the word preceding it in the chain. Examples are:

afraid - apprehensive - expectant - hopeful - confident - unafraid

even - level - uniform - regular - periodic - spasmodic - uneven

valid - convincing - plausible - specious - unsound - invalid

If nothing else, this suggests that synonym substitution in an NLP system must be carefully done.

The other study was made by Amsler [Amsl 80], [Dewd 87]. He recursively rewrote the definitions in dictionary entries in terms of the definitions of the words used in the first definition. The goal was to discover if there was a set of primitives used in all lowest-level definitions and if the number of these primitives would stabilize. He reported that there is a small set of such primitives. They include *food*, *person*, *thing*, *instrument*, and *group*. The nature of these primitives is not like the primitives in many systems: in chemistry, for example, the primitives are protons, electrons and neutrons, and all objects are built from them. The primitives in Amsler's set are mostly class types, which *include* objects of a more specific nature.

What conclusions may be drawn from these observations and studies regarding an NLP system that purports to mimic human processing of language? First, words and their mean-

ings must be able to be categorized in some fashion. The NLP system must understand what words or phrases are close in meaning to others; this is done by humans both by categorization and associations. Second, the definition of words by other words, usually synonyms, has serious limitations and problems. The standard kind of lexicon that has been traditionally used by NLP systems with heavy reliance on synonyms to provide meaning does not represent the way humans know and learn words. As was discovered through the work on UGURU, the wayward synonym problem analyzed by Hardin crops up frequently and in ordinary contexts. The associative memory model of Waltz and Pollack may be the most promising alternative approach to implementing "meaning" for lexical values that has been developed. Rumelhart, McClelland and Hinton [Rume 86] also point out that such a scheme provides inherent categorization.

2.3.2. Parallelism in the Brain

It is not possible to discuss here the vast amount of literature that exists on neuroscience. There appears to be general agreement on several important structural features of the brain, however, which ought to be mentioned since they were influential in the design of this project.

Language processing, and thought processing in general, apparently result from the parallel activities of distinct areas of the brain. Proofs have been offered that are based on the known speed of synaptic connections. Since neurons are relatively slow (compared to computers), the brain could only arrive at answers quickly if problems are tackled by it by dividing up the problem and attacking the parts in parallel. Furthermore, it is known that stored knowledge may be duplicated in several areas.

Marvin Minsky, one of the most prominent contemporary cognitive scientists, believes that these activities of the brain proceed without direction, that is, without the generalship of one particular component of the brain [Mins 87]. In a sense, an idea that can be said to have reached the point of being "thought of," may be some network of neurons stimulated to the

point where it simply dominates other concurrent mental processing. From the perspective of parsing English, this idea of Minsky's can provide the basis for parallel processing of all the distinct parse trees, including those further distinguished by different semantic interpretations. The parse tree that is reinforced the greatest amount emerges as the correct one. This is the approach used in the latest revision of UGURU.

2.3.3. Computer Models Related to Human Language Processing

Most contemporary work on modeling the human thinking process derives from the idea developed by Rosenblatt [Rose 62]. He called his model of a simplified neuron the *perceptron.* There has been much study on this topic since that time without the corresponding production of computer models to match. In fact, Minsky and Papert demonstrated that (one-layer) neural nets have only the power of finite automata. However, increasingly such computer models are appearing, especially since there is more than ever the chance that implementation can be made on a computer with parallel architecture.

Rumelhart and McCleland have been leaders in this field, together with the Parallel Distributed Processing Research Group which they organized. The current version of UGURU has drawn on their models of TLUs (threshold logic units), and of distributed representations of data (in this case, knowledge). The TLU is an enhancement of the perceptron model, and is the basis of much of the study on neural networks. The model of Waltz and Pollack is indebted in several instances to these ideas.

The borrowing in UGURU has been somewhat more in spirit than in specifics. The aspects of TLUs that seem attractive are that each unit is functionally independent of other units, and that the output of any unit reflects some weighting attached to it by that unit. This fits well the ideas of parallelism discussed in the preceding section.

2.4. UC: the Unix Consultant

UC is a natural language interface that provides information on Unix to naive users. Its creators, Wilensky, Arens, and Chin [Wile 84], state that its goal is to help new users "learn operating systems conventions in a relatively painless way. ... UC allows the the user to engage in natural language dialogues with the operating system. The user can: query UC about how to do things in UNIX; ask about command names and formats; receive on-line definitions of UNIX or general operating systems terminology; and get help debugging problems in using UNIX commands."

Many of these objectives are also goals for UGURU. However, the surveys of sample questions gathered for UGURU led to a rethinking of the kind of help that would be generally requested from a Unix consultant, even by "naive" users. Because of the surveys, UGURU has tried to incorporate the facility to provide information pertinent to a wider range of users.

Several of UC's facilities mentioned above are beyond the scope intended for UGURU. For example, UGURU treats questions as isolated inputs, and is not instructed to gather several questions together as part of a conversation. On the other hand, the means are available to store previous questions as part of the cache in the knowledge base, and these may provide contextual aid in parsing a current question. In fact, the description of UC makes it appear that its conversations are of this sort, rather than real conversations, though the knowledge of the previous questions also includes inferential data such as the purpose of the questions.

UGURU also does not have the ability to analyze. Therefore questions that seek debugging help have not been considered as part of its domain. The actual session with UC exhibited as a demonstration of its abilities does not include any questions that illustrate this capability. It is apparently based on modules within both the parsing and knowledge recognition components of UC that provide goal analysis (for the parser) and understanding of plans, or complex concepts (in the case of the knowledge recognizer).

Robert Wilensky has a background of work with the conceptual dependency theories of Schank. Prior to UC, he was the author of PAM [Wile 81]. Much of the theoretical basis for UC comes from this background. The parser is driven by a grammar that is fundamentally a semantic grammar, as are most NLP systems that use elements of conceptual dependency. Knowledge representation is in essence frame and script driven.

The implementation of UC draws on a range of tools that were already developed and tested in other contexts. These include PHRAN, the central parsing element (implemented by Arens), PANDORA, a goal analyzer, PEARL, which manages the data base of Unix knowledge, and PHRED, which generates English versions of the answer.

PHRAN goes beyond semantic grammars by attempting to "develop grammatical categories that are not a function of the particular domain of discourse." Surprisingly, it separates "knowledge about language meaning" from "language-processing strategies." The primary data structure employed by PHRAN is the *pattern-concept* pair. As the input is being read left to right, pattern-concept pairs are activated that may match subsets of the input. Pattern-concept pairs may be deactivated when remaining portions of the pattern fail to appear. Patterns that have been successfully matched are saved, together with the concepts that represent the meaning of the pattern in terms of frames built from the primitive actions of conceptual dependency, such as PTRANS.

PHRAN works only at the sentence level, but interacts with the knowledge manager, PEARL, through the establishment of a large frame-like data structure called the "context model." The context model contains a collection of entries that have associated levels of activation. An idea referenced in a question by the user may become current, and activated into the context model, and subsequently deactivated. Entries in the context model are further organized into clusters that represent situations or associated pieces of knowledge. Any member of a cluster that is reinforced also increases the activation of other members within the cluster. This interesting feature seems related in spirit to the idea of associated memories.

A design goal specified for UC that has also been a goal for UGURU is that the system can be extended in a relatively straightforward way. For this purpose, a component called UC Teacher was added that allows the addition of facts about Unix by stating the fact in English as input. Examples given by the authors show the learning of new vocabulary in this way. An interesting and admirable feature of this learning is that new words are learned by their context, rather than purely by synonym substitution. The new word is saved as part of a PHRAN pattern.

2.5. Conclusion

Of the various theories and models available for further development, the model of Waltz and Pollack and case grammar theory had the greatest influence on the design of UGURU. It drew inspiration from the parallel aspects of the former and the functional treatment of syntax in the latter. Apart from the substantial amount of work done based on the conceptual dependency theories of Schank, there seems to be a growing movement toward implementation of NLP systems that look to parallelism as the solution.

The articles that describe implemented models present sample results generated by the model from inputs that tend to be on the less complicated side. Since this is generally true, it may be assumed it is not merely a simplification aimed at making the article easier to read, but another indication that the general solution of natural language processing is as yet a good ways off. There is still plenty of room for experimentation. In his book *Applied Natural Language Processing* (which is mainly concerned with Schank theory models, actually), Steven Schwartz says [Schw 87]:

> ... few of these systems have more than an extremely fragile natural language interface. Many can only correctly process a small fraction of reasonable natural language inputs.

CHAPTER III

EARLY VERSIONS OF UGURU

Several designs, along with their variations, have been considered and tested for UGURU. Each design was given at least a partial implementation and then abandoned as the difficulties with the design began to suggest that a complete implementation was unlikely to succeed. The earliest, a keyword extraction scheme, planned to use the text of the Unix Users' Manual pages itself as the knowledge base. After this, all the succeeding versions of UGURU included a parsing function for creating correct syntactical parse trees. The second version of UGURU implemented the parsing function with the production rules of a phrase structure grammar. To produce an answer, this version expected to map the parser output to frames representing the different questions that the system knew how to answer. Each frame would contain the location of the actual text of the answer where it appeared in the manual pages.

The second version foundered in the difficulties of producing an adequate phrase structure grammar for English. At this point, the collection of surveys began. Sample questions were requested from the group of people most likely to use the system. The surprising variety and complexity of the questions caused some rethinking of the scale of the system: if it were ever to be actually useful and cover the domain of practical questions indicated by the surveys, the power of the parser and the range of knowledge required considerable extension beyond the first two versions.

The third version began with the parser constructed with ATNs. The limitations of this tool led to the introduction of a stack to be used in combination with the ATNs for the latest possible reduction of the larger elements in a sentence. Though developed independently, this scheme resembles the wait-and-see parser of Marcus. The wait-and-see parser is strongly oriented towards syntactical analysis, and this last version was abandoned in favor of finding a design in which parsing would include concurrent and integrated semantic analysis.

The remainder of this chapter covers the details of these early designs.

3.1. Keyword Extraction

There are several tools available on-line with Unix systems intended to make information about Unix commands available. The text of the Users' Manual itself is on-line; the *man* command exists specifically for calling up the pages of the manual describing the argument supplied to *man*. For novices, the drawback to *man* is that one must already know the name of the command desired and its correct spelling.

The *apropos* command exists to help remedy this situation. *apropos* uses the argument supplied to it as a keyword into the single-line header associated with each command in the manual. It is not strictly a keyword search, in fact, since *apropos* treats the argument as a substring, and pulls out any command that contains the substring in the header line.

The earliest version of UGURU attempted to expand on the basic idea of *apropos*. By extracting the key words from the input question and locating the manual's header lines that contained the keywords, a pool of possible answers would be selected. Further elimination and selection could proceed by keying again, this time into the manual pages themselves with which the name fields chosen were associated. Hopefully, a best choice would emerge from this process. In addition, keying into the itemizations of the command line options usually listed at the beginning of the manual entry might allow an even more specific answer to be given. Also, in as much as the manual pages are frequently lengthy, the answer given could be pared to the most pertinent information, again by keying into the text. In any case, the set of selected manual pages originally hit by the keywords would have to be reconsidered in order to eliminate those altogether inappropriate.

3.1.1. Testing the Keying Scheme on Manual Header Lines

The steps taken towards implementing this idea involved several schemes that tested the probabilities that this kind of keying would be effective. First, a permuted index of all the header lines (in section 1 of the manual) was created. The index listed all words that appeared anywhere in any of the header lines, and mapped each word to a list of the command names that were keyed by that word. With the index available, it was easy to consider sample questions, extract their keywords, and do lookups in the index to determine what commands were keyed as answers to the questions. Out of a large number of questions considered, a pool of approximately forty questions gradually accumulated that were used extensively for testing the effectiveness of the keying procedures described below. The actual extraction of keywords from the questions was always performed "by hand," and no algorithm was ever implemented to produce the keywords automatically. It was felt, however, that intelligent simulation of this function could not be surpassed by a program, nor probably equaled. If the keying tests were not successful, there would be no point in building the extracting routines. The programs that did the actual keying, producing tables of weighted answers and other statistics, were mostly written as Unix shell scripts.

For each of the test questions, many different sets of keywords were produced. Primary sets included the one or two most prominent words; sets including the most prominent words and also significant, but less prominent, words were the next choices. Only words contained in the actual phrasing of the question comprised these primary sets, and they rarely exceeded 6 words. Numerous other sets were generated by the substitution or addition of synonyms for originals in the primary sets already described. Also, the substitution of class names, short phrases, or closely related words created other sets that could be tested. For example, from the question

How can I see what's in a file?

the following sets of keywords emerge naturally: [see], [file], [see, file], [view, file], [read, file], [look, file], [see, in, file], [look, at, file], [contents, file], [see, contents, file], [list, contents, file], [show, file], [view, file, screen], and so forth.

Once a set of keywords representing a given question was generated, it was a straightforward matter to form the set of commands keyed by any of the keywords. When there was more than one keyword, both the union and intersection of the sets of keyed commands were considered. Union and intersection of retrieved sets of objects are standard techniques in such data base query systems as bibliographic retrieval. In this case, the intersection contained the commands keyed the most often through the header line. Since the header lines are short in length, any significant word usually appears only once. Rating the commands was, therefore, a matter of numbers: whatever command was keyed the most times was considered the most likely answer as determined by the given set of keywords..

It had been hoped that this method of selecting the most likely answers would be reliable enough that further use of the keywords would be necessary only to refine this preliminary answer. The results showed that this was not so. About two hundred questions were tested on the basis of just one or two keywords, and it was assumed that these questions represented a reasonable sampling. When using either one or two keywords, the rate of success was about forty percent. Success here means that 1). the command sought was in the "intersection," and 2). only one set of (one or two) keywords extracted from the question

had to work, even though all other keyword sets from the question failed. These particular keyword sets included synonym substitutions, although the synonyms allowed were common substitutes and not obscure ones, such as the substitution of 'catenate' for 'see' in the question "How can I see what's in a file?" mentioned above.

The forty percent was not an encouraging figure, and if the requirements for success were made more stringent, the figure dropped considerably. For example, when the "intersection" was required to contain five or fewer members, the rate of success fell by about ten percent. Pragmatic considerations also diminished the value of the forty percent success rate. About half of the questions that were "answered successfully" were questions on Unix commands that are used relatively little. These questions referenced terms that have a single specific context in Unix, such as "password," "who," "tabs," and "disk." For the same reason it was straightforward to key successfully the commands *mail*, *talk*, *kill*, the commands that do compiling, and a substantial number of others. On the other hand, some major commands, such as *cat* and *make*, were elusive targets to keying by commonly used expressions. The most common problem in successfully keying important commands, such as *ls*, *pwd*, *mv*, and *rm*, was the generality of the words used to identify the command in its header line. Many other commands were keyed simultaneously with these, perhaps even weighted higher, because the words in the header line did not offer enough specificity to distinguish them. As an example, the question

How can I list just the lines in a file beginning with numbers?

contains five possible keywords: *list*, *lines*, *file*, *beginning* and *numbers*. The first three key directly into the header line of the command *look*, and the first four key the command *head*. Although the five together do not key any single header line, they could be taken as a request for the command *cat* with its *-n* option that supplies line numbers at the beginning of the lines when displaying them. Each of these keywords becomes specific only in the context where they are used. In this case, syntactical order is an essential ingredient of that context.

Keying into the header lines could not supply command names to a substantial number of questions, due to the subject matter of the questions. Many questions, natural for beginners, seek not only knowledge of command names, but also how to use symbols and control characters, and how to tailor commands to a particular purpose. These subject areas include redirecting input or output, interrupting running programs, using "history," combining files, and understanding error messages. The information on these subjects (what there is) is embedded in the text of the manual pages, mostly in the sections on *sh* or *csh*. The header lines for *sh* or *csh* do not reflect the variety of topics embraced by these commands, nor could any single header line. So a large block of questions would remain immune to keying into the header lines.

Adding more keywords did not increase the effectiveness of keying into the header lines. In fact it tended to confuse the selection process rather than help it. With the use of keyword sets of three or more words, and the more stringent definition of success, the success rate fell to below twenty-five percent.

3.1.2. Testing Supplementary Variations of the Keying Scheme

The second stage of keying was to apply the keywords to the text of the manual entries for commands selected by the first stage, described above. This could include supplementary keywords generated by the first stage. The second stage was expected to help clarify the results of keying through the header lines. These objectives were:

(1) select a best choice from the commands in the "intersection," if it contained more than one member;

(2) if a significant intersection did not exist, select a best choice from the "union;"

(3) select only appropriate sections of the manual entry for the best choice command, perhaps indicating a particular option to be used.

To test the possibilities of accomplishing these goals, a pool of forty-odd questions was used. For each question, many different keyword sets were prepared. By *grepping* the

manual entry for each command in the "union" of commands keyed by a given keyword set through the header lines, charts of frequencies of the appearance of each keyword in the texts were generated. Thus the charts contained information that showed the effectiveness of keying in solving objectives 1 and 2. Charts were also prepared where the commands *sh* and *csh* were always automatically placed in the "intersection," whether they were keyed or not, to insure that they were there for the numerous instances mentioned above, where the answer to the question asked lay embedded in the text of the manual entries describing the shells.

The charts recorded frequencies. Numbers were less effective here than with the header lines, since the length of the text of manual entries varies greatly. A higher frequency by itself could be misleading if it were due merely to the length of the text. Therefore, some charts were made where the frequency was divided by the number of lines in the entry.

In any case, the results were extremely poor. There was no improvement over the results of keying only through the header lines; the text frequencies in many cases seemed random when considered along with the header line frequencies. Some efforts were made to produce charts by hand showing the ability of keywords to select a particular option for a given command, again by frequency of appearance in the accompanying text. The successes here were negligible.

The other idea developed to bolster the effectiveness of keying was the construction of a thesaurus, which would help to translate words from the question as phrased to words that were known to map to a certain command header line. By working the keying scheme backwards, that is, starting with the header lines as written, it was possible to create a list of keyword sets where each keyword set retrieved one command. The problem of answering a question resolved itself into the problem of mapping the question to the proper set of keywords. The thesaurus was the device by which this would be done.

Once again the results were unpromising. There were many ways in which this idea did not work; most had to do with the multiple meanings, and therefore the multiple synonyms of different meanings, that can be given for most of the common terms. The word *list*, for

example, might have such varied synonyms as

1). see or display

2). output, produce output

3). ordered arrangement of items

4). set, non-ordered arrangement of items

Since all possible synonyms must be investigated if there is no contextual knowledge to help distinguish a particular use, the thesaurus would add considerably to the total keyword sets derived from an input question. Since the synonyms themselves may not have great specificity either, the additional work created cannot guarantee an improvement in results.

3.1.3. Conclusions Drawn from the Keyword Scheme

After the failure of the techniques described above, two other modifications of the keyword scheme were considered. One was to add weights to the entries in the thesaurus that would in some way indicate its degree of specificity and its power for keying effectively. This can be quickly seen to be haphazard at best. Many terms, such as *file*, *system*, or *user*, are generally redundant and carry little information value. Occasionally, however, they may be crucial. This means that the specificity is polarized at two distant values, and cannot adequately be expressed as an average.

A second modification would be to rewrite the header lines so they would be distinctive. This can be done without any knowledge of how the question will be phrased. Each would contain a set of keywords that would uniquely identify it. The function of the extracting process would be to map the actual words used in the question to the set of keywords already known to map to the desired command. In other words, many ways of phrasing the question all map to the same keywords.

Since the number of commands is a relatively small finite number, this approach ought to be possible. The thesaurus might include not only single terms, but sets of terms indicating the most frequent combinations and the targets they are keyed to. This approach ought to

greatly increase the percentage of correctly retrieved commands, but certainly would still fail on many, such as the question, "How can I list the lines in a file beginning with numbers?" discussed above.

The major problem with this approach is that it would be inflexible. Extending it, especially to allow questions that ask for special uses or options for the commands, would be difficult. This was in itself considered an adequate reason for not attempting an implementation along these lines. For this reason, and those summarized below, the keyword attempt was abandoned.

(1) The problem of several meanings or several synonyms of a given word can be resolved only through knowledge of the context or the syntax, and often both.

(2) It is not obvious which are the keywords in a given statement. In different contexts, the same word may supply varying amounts of information.

(3) A knowledge of conjugational and declensional forms is necessary to recognize the different forms a word may take. Closely related to this problem, the knowledge of low level syntactic bindings is also necessary, e.g., to recognize the difference in meaning between *line number* and *number of lines*.

3.2. Phrase Structure Grammar

The failure of the keyword scheme meant that UGURU needed to have a more powerful mechanism for understanding an input question. The second version tried to accomplish this by implementing a phrase structure grammar. The problems that surfaced while working with this implementation partly centered around the limitations of the syntax available in Prolog, and partly the nature of the task itself. It must be admitted that greater knowledge of the problems and possible solutions of natural language processing at this stage would have allowed a more flexible handling of the PSG version. However, the cumbersomeness and a certain rigidity would probably have remained always.

3.2.1. The Production Rules

To facilitate defining a grammar, Prolog has a built-in syntax for production rules. Each rule contains the infix operator -->, whose left-hand side is a nonterminal and right-hand side is the replacement string. For example,

```
noun_phrase --> determiner, adjectives, noun.
```

Any nonterminal is actually a Prolog predicate, and within the --> syntax, each nonterminal represents itself and two implied arguments, each of which is a list of words. One could verify that the beginning of the sentence "A directory may contain subdirectories." is a noun phrase by issuing the goal

```
noun_phrase( [a,directory,may,contain,subdirectories], Rest ).
```

and with the verification, the variable **Rest** will be instantiated to the remainder of the list that follows the noun phrase, presumably [may,contain,subdirectories]. **Rest** could then be supplied as an argument to some other nonterminal, such as verb_phrase.

There are several important characteristics imparted to the parsing process by the nature of this syntax and the nature of Prolog itself. When a nonterminal is invoked as a goal, Prolog executes a depth-first search of all the possible parse trees, expanding the rules in a top-down manner. The order of the rules is crucial to the parse tree produced because Prolog expands the nonterminal with the first rule that it finds that contains the nonterminal as the left-hand side. For UGURU, rules were ordered so that the most frequently used rules were placed first. The main reason for doing so was to increase the likelihood that the first parse tree that was derived was the correct one. Without the introduction of some sort of semantic markers derived from the lexicon (which was not done)[1] to provide a system of constraints among the words themselves, the frequency of ambiguous parses is high. The

[1]Chomsky incorporated this element into his revised standard theory in *Aspects of the Theory of Syntax* [Chom 65].

most common problem would be the proper attachment of prepositional phrases. Since Prolog performs depth-first searching, a complete parse must be produced incorporating one of the possible attachments before correcting the parse can begin. Then backtracking will eventually produce an alternative. By placing the most frequently used rules first, the number of legally syntactic but incorrect parses would hopefully be reduced.

It might be expected that an added benefit of placing the most frequently used rules first would be greater efficiency overall. Actually this did not seem to be the case. Since these rules were often the more complex and led to more involved expansions that had to be tested, a great deal of dead end work was generated. This was taken as the lesser of two evils. An alternative to ordering the rules was to place the less complicated and less expansive rules first, as recommended by some texts on Prolog. These can be decided quickly and eliminated if necessary. There are occasions when this needs to be done in Prolog, however. When recursion can develop, a rule that can end the recursion must be placed ahead of the rule allowing the recursion. In the grammar being used for UGURU, such cases developed a few times due primarily to the use of ϵ-rules. For the sample rule mentioned above, either adjectives must have an ϵ-rule, or an alternative noun_phrase rule must be written. In general, the use of ϵ-rules reduced the number of hard-wired rules that would otherwise have been necessary. Rules for the appearance or non-appearance of commas within noun phrases or separating clauses, and so forth, is an example. In the grammar used, the nonterminal **comma** was defined as follows:

```
comma --> [ ].
comma --> [,].
```

The ϵ-rule version comes first, in this case because of its much higher frequency of use. In other cases, the order had to be negotiated. Nonterminals whose replacements allowed the ellipsis of certain elements were the main culprits.

Another characteristic of this syntax is that the built-in arguments provide no information as to the final parse tree. Therefore, arguments must be added that can be instantiated

as return values. For example, one of the rules for **subj** (subject) became:

```
subj( Path, Elmt, q(D,Ac,N,M,Cs) ) --->
        det_adj( Path, Elmt, D ),
        adj_cls( Path, Elmt, Ac ),
        ntype( Path, Elmt, N ),
        post_mod( Path, Elmt, M ),
        cjsubj( Path, Elmt, Cs ).
```

In order to make correct choices, it began to seem necessary also to pass arguments down from the top, so that a nonterminal expanded in a subordinate clause might be treated differently than its expansion in a main clause. The variable **Path** in the above rule carried this kind of information. In the question "Can I change a file while printing it?" the "I am" is omitted from the dependent clause. In general, an ϵ-rule for the subject and auxiliary verb would be undesirable, but if the path is known to have led to a conditional clause, such a rule can be considered.

Information from the **path** variables was also used to formulate the *frame* entries created when terminal symbols were encountered. Each terminal was assigned its own frame predicate, and the entire group of frame entries constituted the parse tree and was returned to the top level of parsing calls. A frame entry contained the path to the terminal, a weight that gave a very low-level indication of the importance of the terminal, and the terminal itself. The weight system was fairly superficial and was only used to eliminate the most obviously trivial words in an effort at efficiency. It assigned a higher weight to the less crucial terms. Had there not been greater problems with the writing of the production rules, the weight system certainly would have been expanded. Figure 3.1 shows the frame produced that represents the parse tree for the question "How can I remove square brackets from a line without deleting it?" There are obviously problems that were left unsolved. The automatic *un*weighting of "it" ignores the problem that some resolution of the reference must still be made. A more serious problem is that the path identifiers could become extended and very

```
fr( vb_md(main), 1, how ).
fr( int, 1, how ).
fr( vb(main), 21, can ).
fr( subj(main), 21, i ).
fr( vb(main), 1, remove ).
fr( obj_md(main), 1, square ).
fr( obj(main), 1, brackets ).
fr( vb_md(main), 3, from ).
fr( obj_md(obj(vb_md(main))), 21, a ).
fr( obj(vb_md(main)), 1, line ).
fr( cond(cond), 1, without ).
fr( vb(cond), 1, delete ).
fr( obj(cond), 21, it ).
```

KEY

vb_md(X)	verbal modifier in clause X
int	interrogative specifier
vb(X)	verb of clause X
subj(X)	subject of clause X
obj_md(X)	object modifier in clause or object X
obj(X)	object of clause or object X
cond(X)	conditional conjunction

Fig. 3.1 Parser frame for the question
How can I remove square brackets from a line without deleting it?

awkward to work with. The information they provided was purely syntactic, and much less than was necessary for understanding the inner relationships of the objects referred to.

3.2.2. Approach to the Problems of English

Since the domain of input to the parser was questions about Unix commands, certain assumptions were made about what would be and what would not be necessary to include in the production rules. First of all, each input was to be a question. The production rules, which grew to one hundred, correspondingly did not include rules for statements. Although some rules handled some forms of ellipsis, mostly relating to subordinate clauses, the parser

was not expected to handle partial sentences (or questions). These usually arise in a conversational context, and the missing sentence elements can be filled in from previous statements. The questions UGURU was being designed to meet might contain conditional clauses, or other subordinate clauses, but each question needed to be complete in itself.

The vocabulary was set up through declarations of Prolog predicates. The entries contained syntactic information only: the word itself, and its part of speech. For example,

noun(terminal).

Because Prolog performs the lookup of predicates much faster than a stem-producing procedure can execute, separate entries existed for singular and plural nouns, inflective forms of verbs, and comparative forms of adjectives and adverbs. The gain in execution speed outweighed the cost of the duplicate allocations. The format of the entries conformed to the requirements of Prolog's production rule syntax. A rule seeking to match the nonterminal **noun** with an argument list **[terminal,...]** can execute directly. The lexicon did not clarify the problem of words like "command," which have multiple syntactic functions. Two otherwise unrelated entries were made for "command," one as a noun, the other as a verb. The system relied on the overall syntactic constraints in a sentence to disallow the wrong use of such words. For the question

How do I save all the output mistakes from my program?

there is also the legitimate (syntactic) parse that treats "the output mistakes from my program" as a relative clause, with an elliptical "that," that modifies all. In other words,

How do I save all [that] the output mistakes from my program?

Generating a correct parse for this question, without using semantic knowledge about the abilities of "output" to make such observations, depended on the ordering of the rules. The rules that treated "all the output mistakes" as a noun phrase had to proceed those rules that parsed elliptical relative clauses. On the other hand, the problem was complicated by

occasions when the elliptical relative clause rules had to proceed the noun phrase rules.

The noun phrase rules that would recognize "all the output mistakes" correctly require subordinate rules that recognize compound nouns. This was handled in a somewhat rigid manner, but it is not an easy problem to solve, of course. The compound noun rule recognized a combination of nouns if one noun was an example or component of the other, the most frequently appearing pairs in this domain being a combination of the word "command" together with an example of a Unix command, or possibly a verb ("What is the print command?"). A small extension of the lexicon included a list of nouns and examples or components of these nouns, and additions were made on an ad hoc basis as they were needed to effect a proper parse.

The proper attachment of prepositional phrases depended on procedures that executed following the production of a parse tree for the input. For making some sort of judgment, the grammar rules used a principle of *proximity* for attachment: usually a prepositional phrase immediately follows what it modifies. When this is not true, usually the phrase modifies a verb, but there is an intervening noun phrase, perhaps the object of the verb. The proximity rule allowed the grammar to make a reasonable guess, and if the checking procedures disallowed the guess, they caused backtracking and the production of an alternative parse tree. An extension of the lexicon was added for prepositions, much like the extension for compound nouns. These lexicon entries gave information about the acceptable use of each preposition, frequently limited to parts of speech it might or might not be combined with. The checking procedures used the information to deny attachments that were clearly wrong, which meant that many wrong parses could slip by. This arrangement was quite coarse-grained, but on the other hand, a fairly high percent of prepositional phrases were parsed correctly.

Rules to handle conjunctions were created by extending the basic rules that recognized sentence constituents. The rule for the nonterminal **subj** discussed above is an example. The replacement string contains nonterminals for **det_adj** (a determiner), **adj_cls** (an adjective

clause), **ntype** (a noun or noun substitute), and **post_mod** (trailing modifiers, such as prepositional phrases). The nonterminal **cjsubj** (conjunctive subject) is defined with the following production rules:

```
cjsubj( _, _, fr([]) ) --> [ ].
cjsubj( Path, Elmt, q(Cj,S)) -->
        cj( Path, Elmt_, Cj ),
        subj( Path, Elmt_, S ).
```

The first rule is the necessary ϵ-rule for the majority of cases where the subject is not conjunctive. It is also necessary to satisfactorily end recursive calls to **cjsubj** without a failure (Prolog predicates either succeed or fail). Similar conjunctive nonterminals were added to clauses for **verb** and **pred** (predicate). Like the attachment of prepositional phrases, this arrangement worked with fair consistency, but most sample inputs did not test it severely.

The production rules neither tested for agreement of subject and verb, nor marked the tense of the verbs. Since the input domain would comprise mostly questions of the sort "How can i ..." or "What is ...", the present tense would predominate. With conditional clauses that included past tense, the conditional aspect of the clause would generally (though not always) imply a past tense anyway. These shortcuts were allowed for the sake of efficiency, and were carried over into the subsequent versions of UGURU. Preprocessing of the input also meant less burden was placed on the production rules. Preprocessing included the rewriting of contractions and the merging of many idioms into single compound words that could be found in the lexicon in the compound form.

3.2.3. Conclusions Drawn from the PSG Version

The problems of working with the PSG implementation in Prolog were considerable, even though the kinds of questions anticipated used relatively simple language.

One set of problems centered on the depth-first search of the parse trees executed by Prolog. How well the parser worked was a function of the order of the rules as well as of all

its other elements. As the order became more rigid with the addition of new rules, the difficulty of adding more new rules also increased.

Two aspects of the grammar made it exceedingly difficult to judge the effect of adding a new rule. There was little transparency as there is in the WASP parser rules, for instance. One of these aspects was that each nonterminal had numerous replacement strings. As many as six was not uncommon. As calls to the rules could be nexted to many levels, the number of permutations of replacements chosen becomes very large. The other difficult aspect was that the order in which the rules were written determined which replacement string was chosen. Adjustments made to rule order might solve one problem, while undoing those already thought to be solved. Furthermore, since the grammar was started from scratch, the implementation of rule syntax and data structures was constantly changing. For all these reasons, rewriting one rule seemed to entail rewriting many more others, and this was a major source of discouragement.

Another set of problems concerned the naturalness of the method. A major point to be made is that rules are not written for expressions that are incorrect syntactically. From a practical point of view, the incorrect possibilities are limitless, and probably more difficult to characterize than the correct possibilities. The same is true for writing rules to cover all cases where ellipsis can occur. Yet people process awkward, incorrect or incomplete language on a regular basis. Related to this is another troubling aspect, discussed by Harris [Harr 85]. A set of production rules can generate immensely more syntactic but nonsensical sentences than sensible ones.

Backtracking also does not seem to be a compatible ingredient in an NLP system when one considers that people are the original natural language processors. Even if the semantic checking procedures in this version of UGURU had been fine-grained instead of coarse-grained, there is still a point where comparison of two alternative parses seems necessary. Consider the two sentences:

Putting a sign on the bench *earlier* would have saved some people from ruining their

clothes.

Putting a sign on the bench *probably* would have saved some people from ruining their clothes.

Earlier modifies *putting* in the first sentence, and *probably* modifies *saved* in the second. If the rule that assigns an adverb in such a position to the main verb of the main clause comes first, the checking procedures would see a parse tree for the first sentence that attaches *earlier* to *saved*. This is not an implausible construction and certainly not incorrect, but it does not convey accurately the meaning of the sentence as we know it. The checking procedures would be hard put to find something wrong with it. If mere doubts could be expressed by the checking procedures, they could hold on to the first possibility and force production of other possibilities. This is not worthwhile because it cannot be known when a better possibility would come along, and so all possibilities would have to be generated. In this case, it is better to try to come to a decision sooner and avoid backtracking.

3.3. The Question Surveys

The problem of retrieving an answer was yet to be solved in a general way, and only questions phrased just the right way produced frames that matched the stored frames, which represented answers. However, the PSG version of UGURU was capable of parsing most of the questions fed to it. This could be guaranteed by making adjustments that would cover new input questions, although the difficulties of adding new rules or rewriting rules seemed to place limitations on this. A large set of sample questions that represented the real world was needed to determine if the PSG version was a dead end.

Students in the surveys were instructed to recall actual situations where they felt they had needed to ask questions about Unix. They were to write down the *questions* they would like to have asked. The surveys revealed the following: "simple" questions were in the minority. The questions were less than simple for many reasons. One reason is that often very specific or qualified information about even the most basic Unix commands was sought.

For example,

How can I edit every source file that contains the constant MAX?

Providing answers to many questions would involve guessing about some missing assumptions in the question. Letting the system give several alternative answers when it could not decide on one was possible, though not preferable if that situation could be avoided. For example,

How do I move a file inside another file?

The final relationship of the files is not specified. A simple "cat" with the >> redirection symbol would at least be an answer that allows the two end-to-end possibilities, but it is likely that what this student is looking for is a more flexible answer than that. It may very well be that this student has neglected to say that he is already at work inside one of the editors, and needs the line-editing command *r*.

Many questions concerned themselves with finding a better way of doing something that added a sort of negative logic into the problem of finding an answer. These questions were frequently formatted along the lines of "how can I do ..., but not ...?"

How can I search all my files for a word without opening the files separately?

How do you reexecute some previous command with a substitution of some sort, so you don't have to retype the whole thing?

The use of the English language in complex ways, or non-standard ways, even in questions about simple Unix features, is what most of all made the questions not "simple." Clearly a system like UGURU needs to find a solution to misspellings and ungrammatical statements. It was also apparent that some of the questions were from foreign students. The eagerness with which people use special punctuation to make themselves clear was also surprising. Approximately ten percent of the questions used parentheses or single or double quotes. The students wished to highlight an item they apparently regarded as unusual or unfamiliar, or

they might provide an example that was quoted. The combination *and/or* was also relatively popular, as was the use of abbreviations. Indicative of these is the following example:

> I have this message that says "You have mail"; now what do I do?

The question asks for straightforward information about a simple Unix command, but is formulated in a way that presents substantial difficulties in parsing.

The last example also illustrates another problem, at least as far as UGURU might view it. Although the students were specifically requested to formulate questions, approximately fifteen percent of the responses were either in the form of a statement, contained a statement, or were incomplete sentences. Few were difficult to understand upon reading them, but if they were allowed into the domain of input to UGURU's parser, the complexity of the parsing problem grew substantially.

Students who might not be familiar with standard terminology might use words in an odd way, as in the following use of the word *branch*:

> How do you call up, run, programs that are in different branch then [sic] what you are in?

Or some tended to add redundant phrases, or a large number of subordinate clauses:

> I would like to be able to know how to print out the program and the execution statement, like the grader does.

> If a file is longer than I can save on the disk, what can I do so that I don't have to get rid of it?

> At my coop, we were able to create an xedit file which could be run against a file(s), executing all the edit commands against the file(s). (Sort of like a grep with a substitution if found.) Is this possible on unix, as my coop was with an IBM environment?

A large number of other variations were observed in the surveys as well, where each variation occurred infrequently. This was disturbing, since it could mean that a workable set of

allowed questions would be so small that no practical value could be imagined for the system. These other variations seemed to be hard to categorize. A few students tried to address the program directly.

Hey, guru, how can I permanently change the Unix prompt?

Excuse me, what is the command for repeating corrections in a file?

Another consideration was how UGURU should respond to certain questions that were not relevant and those that were perhaps meant to be rhetorical. The safest way out would probably be to quietly fail to answer. The larger problem was recognizing the questions for what they were.

The set of questions gathered have to be considered the natural ones that people would give if they had a Unix guru available to them. The examples of questions noted above are concerned with the most basic Unix functions, printing files, editing files, locating files, mail, and so forth. The answers do not involve complicated uses of Unix commands. A system that intends to answer questions about Unix should attempt to answer these questions.

It has to be admitted that the surveys were a surprise. They raised the difficulty of the project enormously. From this point, the design for UGURU became more important than the implementation since a complete implementation for the wide range of questions seemed out of reach. It was important that the design be extensible, and in two ways at least. Certainly it should be easy to add new pieces of information to the knowledge base. It was just as important that UGURU's knowledge of English could be extended easily as well.

3.4. Working With ATNs

After the PSG version, two partial implementations of UGURU were made using ATNs. There are probably more NLP systems based on this technique than any other [Ober 87]. It has the advantage that it is comparatively simple to use. Its code can be written to be easily read and understood, and the modularization of different grammatical structures as

separate ATNs makes adjustments and additions somewhat less difficult than with a phrase structure grammar. Through the use of registers, the capability of the system to deal with problems of context can also be increased over that of a PSG version.

There are problems and limitations, too. The grammar represented still becomes enormously complex if a wide range of sentence structures are allowed. ATNs cannot solve in a simple way the problems of displaced modifiers and appositives or embedded clauses. In normal discourse, these are frequently encountered. Another problem concerns conjunctions. In the sentence beginning "I printed the file and the letter ..." the function of *the letter* cannot be determined until more of the sentence has been seen. The technique of cascaded ATNs has been developed to help solve this problem, but has the disadvantage of a high degree of complexity. ATNs also have the disadvantage, common to all parsing techniques strongly oriented around syntax analysis, that there is no easy way to provide for the parsing of statements with incorrect or unorthodox syntax.

Since the use of ATNS is well-known, a detailed description of the implementations will not be given. Instead, the reasons for abandoning it will be discussed, since these led to the development of the parsing techniques used in the current version of UGURU. The problem of knowledge representation was not confronted during the experimentation with ATNs, since work did not proceed past the stage of developing a parser. In general, the goal was to build an ATN parser that produced frame representations of the input similar to the type of frames used in the PSG version of the parser.

3.4.1. First Version with ATNs

This version followed the standard approach for using ATNs. Lower level syntactic units - noun phrases, verb phrases, and prepositional phrases - were recognized by ATN networks. Higher level units - subjects, objects, predicates, independent modifiers - were generally stored in registers. In Prolog, the ATNs were implemented as predicates with names such as **getNPHR** (get noun phrase), **getINFV** (get infinitive phrase), etc. A generalized top

level network with two major states was created to drive the phrase gathering units. The two states were arbitrarily called s0 and s00, and were essentially start and end states. Both states had numerous arcs that looped back onto themselves. Between the two states were three alternative states through at least one of which the parser needed to pass before arriving at the end state. The internal states required approach by way of a noun phrase, possibly preceded by an auxiliary verb, and could be left via recognition of a verb phrase. Objects and modifiers trailing the main verb were recognized within the end state.

The ATN parser had difficulties with the same types of input as the PSG parser. The most frequent source of trouble was the proper attachment of independent modifiers, that is, modifying words or phrases that do not directly precede their object. The most common, discussed earlier in Section 3.2.2, is prepositional phrases that trail verbs. Even though the registers store previously identified elements, they cannot represent the sequence in which these elements appeared. Often this information can help decide how to attach the modifier: frequently it is the most recent.

A second major source of trouble was deciphering subordinate clauses. It seems theoretically correct[2] to treat a subordinate clause as an embedded sentence. The implication is that parsing should proceed in a recursive manner, treating the clause with a progression from its own start to end states. The ATN registers do not lend themselves well to implementing this multi-layered view of language.

Finally, an ATN parser requires backtracking for the same reasons as the PSG parser. Backtracking is the only solution to the initial misidentification of syntactic elements. Based on the argument that parsing can and should be psychologically realistic, backtracking is generally an undesirable feature of the process. Specific arguments were voiced in Section 3.2.3.

[2]The reference here is to Chomsky. See Section 2.1.1.

3.4.2. Second Version with ATNs

Subsequently, a design and partial implementation was made on a modified version of the ATN model that sought to solve the problems discussed in the preceding section. The new ingredient was a stack to replace the registers. This idea is quite similar to the Wait-and-See Parser.

The use of a stack seemed likely to help resolve these problems in the following ways. As with the Wait-and-See Parser, backtracking can usually be avoided since any constituent can be stored on the stack and identified only when the identification can be made certain. Initial guesses do not have to be made. Secondly, the stack records the order of the appearance of the constituents. In many cases, this provides a dependable solution to the proper attachment of independent modifiers and other elements connected by conjunctions. In the design for this version of UGURU, the top element of the stack (a linked list in Prolog) was also allowed to be a stack. In this way, even though this had its clumsy aspects, the embedded nature of subordinate clauses was realized in the data structures used for parsing.

This time the lower level ATNs for recognizing phrases were driven by Prolog predicates called **getMODUL** and **getCELL**. These were generic in nature, and could recursively call one another. **getMODUL** had the task of recognizing clauses, either the main clause or subordinate clauses. **getCELL** had the task of recognizing elements within clauses, which includes embedded clauses. Since **getMODUL** and **getCELL** were generic, they did not have to make initial guesses as to the high level identity of the incoming constituent, as otherwise would be required in top-down parsing.

The predicates that controlled the stack were called **reset**. This set of predicates performed the major identification and attachment of sentence elements. If necessary, these predicates could look down into the stack an arbitrary depth below the stack.

The central problem that arose with this scheme was the coding of the high level predicates: **getMODUL**, **getCELL**, and **reset**. They still implemented a grammar, like the earlier ATN and PSG versions, in which all alternative orderings of constituents at the highest level

in the sentence had to be spelled out. The high level structure of language was still being processed like the lower level elements that are more rigorously ordered. In examining many styles of written and spoken English, including the questions gathered in the surveys, the conclusion was drawn that trying to create a road map that shows all the possible paths a sentence can travel does not reflect the way language works. High level elements are processed in a different manner than lower level elements. This is what allows humans to understand malformed sentences. Although it is believed that a reasonable system could have been written using the implementation coupling ATNs with a stack, the complexity of coding involved suggested that the effort might be better spent trying to develop an approach that was more faithful to the flexible way in which language is used.

CHAPTER IV

THE PARSER:
CONCEPTUAL DEVELOPMENT

The factors that shaped the present version of the parser and the grammar it defines fall into two categories. One is the experiences of writing the early versions of UGURU, which were described in that last chapter. The other concerns the determination to mimic human language processing as much as possible.

Many considerations that resulted from working with the earlier versions were negative responses to them. One conclusion drawn from these experiences was that a considerable amount of semantic knowledge, minimally in the form of word constraints, is necessary to clarify syntactic ambiguities. An equally important conclusion was that the methods used, a phrase structure grammar and then ATNs, seem to be artificial approaches to the problem and probably distort the way natural language processing is *really* done, that is, by human beings. This is particularly true if the parsing process uses one of these methods exclusively.

If these methods were not to be used, another was needed. By trying to match the cumulative, left-to-right processing done by humans, the new approach might avoid some of the problems present in the techniques already tried. The problem here is that we do not know how people process language; however, there are many studies that provide evidence for partial answers. There are also models that display characteristics of human language processing. Chapter II surveyed some of those that influenced this project.

On the other hand, it was easy to develop a simple pragmatic criterion to project how the parser should work, even without reference to these partial answers and models. When a person reads only a portion of a sentence, a certain amount of coherent information has already been gleaned: the parser should be designed so that it can reconstruct as much of the same information with the same portion of the sentence presented to it. This means not only too little of the same information, but also too much, perhaps resulting from jumping to conclusions that prove wrong and cause backtracking. This criterion is simply a heuristic, and may circumvent genuine analysis of mental processes, ignoring large amounts of information that are involved, perhaps even subconsciously. Its advantages are that it is easy to use and that it does not require an algorithmic implementation, but can be hand simulated with self-tests. It was hoped that this criterion would show where some of the missing pieces were with the PSG and ATN methods.

4.1. Use of the Pragmatic Criterion

The use of the criterion can be demonstrated with the following question from the surveys.

> When editing a file in Unix, how can I globally substitute all the occurrences of a variable within a file with another variable?

After input of the first word *When*, at least three possibilities must be considered by the system. *When* is the key interrogative adverb in a question, e.g.,

When can I login?

It may be the introductory word to an adverbial clause used as a noun.

When the system will be up has not been determined.

Or, as we know from reading the rest of the sentence, it is the beginning of an adverbial clause[1] used as an adverb.

This ignores other minor possibilities. There is, for example, always the case where a word is used symbolically.

When is an adverb.

This last situation is not so obscure as it may first seem. Examples like the following occurred in the survey of questions.

How can I move a line from point a to point b in my file?

This example means that the temptation to hardwire *a* as a determiner must be resisted enough to include its possibilities as a noun.

Returning to the example sentence, with only the word *When* visible the system should be aware, as we are, that the second possibility is less likely to be the actual continuation than the first or third. Without having the context of a conversation in which the question has arisen, the only reason for saying so is a comparison of frequency of use. In general, we would much more frequently start a sentence (question) with *When* and continue as in the first possibility than either the second or third. Of course, determination of frequency depends on the domain of sentences being considered. In the surveys, the third possibility is actually more common than the first.

[1]UGURU's notational scheme designates this clause as a *conditional clause*.

The scanning of the next word *editing* does not resolve which of the possibilities should be continued, but actually multiplies them, especially if one considers syntax only. *editing* may be a noun, or part of a compound noun.

When editing a file is interrupted, can I save my changes?[2]

When editing commands are used in "vi" is not clear to me.

Commonly, present participles serve also as adjectives. In this case, *editing* is the central verb in a conditional clause with an elliptical subject (*I*).

The system should be able to eliminate the first possibility when only the first word was known on the principle that the main verb to come would not be separated from the *When* by any phrase beginning with a gerund. Any such case requires the use of a comma to separate *When* from the intervening phrase. Semantic knowledge of the uses of the word *edit* in the given domain also should allow it to disregard the possibility that *editing* is being used as an adjective. The system should also be aware at this point that the likeliest interpretation is the last one mentioned in the last paragraph, which will turn out to be the appropriate one.

The system should assimilate *a file* as the object of editing immediately, whether it knows if the *When*-clause is being used adverbially or as a noun. Semantic recognition that there is a common binding between *edit* and *file* that relates them as action and object should make this clear. Syntax cannot do so, at this point, since sentences such as the following cannot be overlooked:

When editing the student became very frustrated.

Usually (for UGURU, just a matter of higher frequency) a comma would be used in the last sentence to clarify it:

---

[2] *editing a file* is considered to be a *verbal clause* used as a noun in this case. UGURU interprets any verb (or verbs joined by conjunctions) as the focal point of a DAG, a separate semantic net of distributed information.

When editing, the student became very frustrated.

but this is not absolutely required, and both cases must remain options to the parser.

When the first four words of the sentence are known to the parsing element, it should recognize that it is in the middle of an elliptical adverbial clause, as a person does. Note that it cannot be guaranteed that this clause is contained in a question, however, since it could just as well be contained in a sentence (see the last example, above). Assuming it to be part of a question is justifiable in the context of UGURU's purpose, answering questions. The surveys contained, however, numerous examples where statements and questions were combined.

The remainder of the parsing proceeds with fewer complications as far as identifying verbs, subjects, and predicates. The problem of the bindings of the last prepositional phrases is not easy. The "default" solution of binding them to the immediately preceding noun or verb works for *of a variable* and *within a file*, but certainly a more dependable mechanism is required. The binding of *with another variable* requires the system to know that *substitute* is an action that "likes" two objects, an "original" and a "replacement." Since *substitute* can handle this two ways,

substitute x with y

or

substitute y for x

the semantic information about the use of *substitute* must make this clear. The possibility that *with* may begin a prepositional phrase that should bind to *file* instead of *substitute*, as in the question,

How can I substitute all the occurrences of a variable within a file with lower case spellings with another variable?

seems to require semantic knowledge that the two objects of *substitute* have some important common or parallel characteristic. Also, it would be helpful for the system to recognize how many files and variables are referred to: one file and two variables.

It is not assumed that these considerations on the sample question present any startling revelations about the problems of "parsing" English. They do show the considerations that went into the design of UGURU's parser with the incorporation of the mind-matching criterion. After the study of a great many sentences, taken piece by piece, it does not appear that many lead unalterably down a syntactic path, or even semantic path, until close to the end. There are usually several forks in the road at each step along the way. Since it is also clear that out of all sense-making sentences, relatively few are actually garden paths, it is the steady accumulation of various types of information during a left-right parsing of language that is the key to doing it "naturally" and probably correctly.

4.2. Goals in the Parser Design

Parsing, in strict terms, means generating a complete map of the syntactic relations and dependencies of a given input, normally in the form of a derivation tree. In UGURU, the parser produces DAGs in which each edge represents a *binding* that it finds in the input. Bindings frequently represent the *deep cases* of a case grammar, and otherwise they extend the notions of what constitutes a case. The current knowledge of the parser regarding an input, or *any portion of the input*, comprises the set of bindings constructed, and the amount of stored knowledge activated to construct the bindings, either procedural or declarative, syntactic or semantic. Since it is not possible to assess all the knowledge involved in human language processing, the pragmatic criterion has been applied specifically to the set of bindings: if these match the set formed by a person over the same input or portion of that input, the parser is judged to be adequately successful on that input.

Modeling the parser on human language processing means the incorporation of several elements generally agreed to be central to the way the mind works. The parser can be

expected to have elements of parallelism; the structures of the representation of knowledge can be expected to allow a distributed representation of "ideas;" and the manipulation of "ideas" can be expected to involve the use of classes and hierarchies of those ideas.

The parser should avoid the "unnatural" aspects of the methods used in the earlier versions. In particular, it should avoid backtracking. Backtracking does not seem, with the rare exception of true garden path statements, to play a part in "natural" language processing. Also, the parser should not be cornered into relying excessively on syntactic restraints to produce the proper bindings. Even though syntactic restraints are sufficient to resolve a very large share of ambiguities that occur early in a sentence, there are two problems with this reliance. One is that it usually involves backtracking to recover; people would have resolved the same ambiguity earlier with semantic means. The second problem is that the system can reject only those parse trees that reach an "absolute" contradiction; what is needed is the ability to reject those parse trees that reach a "threshold" of contradiction, and such a system does not have this kind of knowledge. Furthermore, it will reject badly formed sentences that may indeed make sense. It will also have trouble handling arbitrary elliptic phrases. Teaching "correct" English is not one of the goals for an NLP parser.

We can summarize by pinpointing two major goals for the parser design. One is to implement a grammar with the flexibility to allow decision-making by various kinds of elements, flexible enough so that one element can compensate for a failure in another element (such as lack of knowledge, or incorrect or awkward syntax). UGURU's parser was developed as a system where syntactic knowledge is used to build up potential parses from which procedures with semantic knowledge can choose or extend even further. The second goal regards the building of the system itself and its extensibility. In view of the experiences of working on the early versions, the grammar itself must have the flexibility to allow additional rules and declarative knowledge to be added easily, without causing the general rewriting of existing rules and knowledge.

4.3. The Viewpoint Taken on the Constructs of English

As its primary function, the parser establishes the bindings that exist between the constituents in the input sentence. This section considers the kinds of bindings and the kinds of constituents that this parser understands, along with the relationship of English syntax to these elements.

4.3.1. Bindings

Bindings associate two words together with an identifier representing the relationship between the two words. In the implementation, the words also carry lexical information, and the relationship may be identified by a combination of terms. Throughout the text, the term binding will be used sometimes to represent the entire triple, and sometimes it will just refer to the type of relationship itself. The context should make clear which use is intended.

Bindings may assert relationships between noun objects and verbs, such as **auth** (author),[3] **obj** (object), or **indobj** (indirect object, without the explicit use of a preposition, such as *to* or *for*). Another important binding that joins a verb and a clause acting as a noun is **purp** (purpose or result of the action). Verbs that relate two noun objects, such as *substitute* in the example in section 4.1, or many others, like *change* or *move*, may generate bindings called **src** (source) and **dest** (destination). Bindings may assert adverbial relations, such as **loc** (location where), **when** (time when), **manner** (answering the question how), **means** (by means of), **near** (nearness to), or **cond** (a conditional situation). Nouns, or constituents used as nouns, may relate to each other with bindings such as **instof** ("is an instance of," frequently the mechanism for binding two nouns as a single compound noun), **cmptof** ("is a component of," also used sometimes to bind compound nouns), or **own** (ownership). Nouns or adjective clauses that are used as independent modifiers of other nouns are considered to have one of the following relationships. A predicate noun, that is, one that follows a state-of-being verb,

[3] **Subject** would be a misleading name, since subjects may instigate or receive the action of the verb. Also, **agent** would be ambiguous, since it frequently refers to the means by which the subject executed the action.

modifies the subject as its **trnsfl** (transferral). Appositives are connected to the modified noun with the relation **appos**. Many relationships are considered **adjmod(s)** or **advmod(s)**, simple adjectival or adverbial modification of their respective objects. The type of relationship can be specified in an ad hoc manner by a given rule if the classification seems to warrant it.

The parser must occasionally assert the negation of a binding. It achieves this by creating two bindings, the original in its positive form, and the second using the **neg** primitive to bind *not* to one of the objects involved. For example, for the question

How can I repeat a command without retyping all of it?

retype is the **cond** of *repeat*, and *not* is the **neg** of *retype*. **neg** can also appear as a half-binding relation, where the specific negative word is absent, but the instance of negation is still tied to the particular object. The logical conjunctions are treated also as half-binding relations.

It should be noted at this point, that the set of relationships that represent the range of bindings are the only primitives in this system. The parser seeks to define a relationship between two constituents in terms of the primitive values when possible. The advantages of doing so seem to outweigh the disadvantages. Even with the wide range of questions drawn from the surveys, the set of primitives described above serves quite well as far as covering the domain. However, a couple of examples will illustrate both the way the program uses it and the nature of its limitations, limitations that are arguably inherent in any system that uses words as primitives.[4]

How can I look through a file without editing it?

File must be bound to *look*. The choice of binding hinges on the idiomatic combination of

[4]This situation is perhaps argument enough in itself to conclude that other means than words must be used as primitive representations. Chapter VIII extends the discussion of this point.

look and *through*: while **loc** is the best choice in this case, **obj** would also work, and possibly **dest**. If the question is elaborated:

How can I look through a file for a name I forgot without editing it?

the addition of *for a name* casts the binding of *file* in another light. **loc** is more clearly the best choice, but **src** rather than **dest** now seems to be an alternative, and **obj** more accurately relates *name* to *look*. Selecting **loc** seems to solve the problem. The same solution must be used for the following sentence.

Every day, he looks through the window for the postman.

window will be bound as the **loc** of *looks*. Though there are now identical bindings between *look* and *file*, and *look* and *window*, the relationships are not identical because the meanings are different.

As a second example, consider the following two questions.

If I accidentally destroy a file, how can I recover it?

When I have accidentally destroyed a file, how can I recover it?

The meanings of the two questions are identical, and the parser should connect *destroy* to *recover*. The first question implies that the binding between the two is **cond**, but the second question implies that it is **when**. We commonly accept that an idea may expressed in more than one way be rewording it, but multiple primitive representations present computational difficulties. In this case, UGURU's parser treats clauses that begin with *when* as conditional clauses that modify the main verb. Exceptions must be handled on an ad hoc basis through semantic mechanisms.

The parser employs a principle that a verb (or any other constituent) will have only one instance of a particular binding relationship connecting it to a dependent or dependents. If a

single object has more than one dependent with the same relationship, those dependents must be joined by conjunctions. This follows normal English usage. One reason to have both **means** and **manner** is that there should not be two bindings of the same sort. In the sentence

... in a hurry with the hammer

there are both means and manner.

4.3.2. Syntactic Constituents

The question of what is a constituent is a very open-ended one, from the evidence of language use. A strong characteristic of English, that may in fact be much less true of other European languages, is the capacity to use one sort of syntactic object as another. Sometimes this simply means using one part of speech as another:

A page fault causes a disk read to be performed.

Here the verb *read* serves as a noun.

How can I see who's on the system right now?

This question treats the adjective *right* as an adverb. Many of these substitutions are so commonly employed that any dictionary gives both uses as alternatives. The question of substitutions is more general than just the substitutions of of one part of speech for another, though, since whole phrases or clauses may substitute for a single part of speech.

How can I trace an out of bounds subscript in the debugger?

The prepositional phrase *out of bounds* creates no problems in understanding when used as an adjective, for example. The substitutions for noun objects seem to have the greatest latitude: a variety of compound formulas, elliptical noun phrases that contain only adjectives (e.g., *the good and the bad*), and all types of clauses, relative, interrogative, infinitive, and gerundive.

4.3.2.1. Levels of Syntactic Constituents

The grammar embodied by the parser recognizes syntactic elements in four different levels. From highest to lowest these are called the *clause*, *object*, *phrase*, and *token* levels. In general, elements belonging to one level combine to form an element of the next higher level. Figure 4.1 lists definitions of the highest level constituents in terms of phrase structure production rules. Instead of the Prolog operator -->, the definitions use the symbol ~~> to indicate that the order and number of elements in the replacement string are not defined in the rule. The replacement string only indicates what lower level constituents may be present. The symbol :: indicates that the right-hand side is one of the varieties of a given type of constituent. The grammar production rules for *phrases*, not listed in Figure 4.1, could be written with the use of -->, since their very definitions constrain the order and number of subelements. They are not listed since they follow conventional ideas closely. When they are used in a replacement string following ~~>, the terms *clause* and *object* should be understood to represent either a single element of that type, or several joined by conjunctions. It may also be observed from Figure 4.1 that, since clauses of various types may substitute as nominatives or modifiers that exist at the *object* level, the definition of an element is potentially recursive. The primary example is a clause that contains a subordinate clause. Parsing the subordinate clause also means determining which of the *object* types it is functioning as.

The lowest level constituent is the *token.* A *token* is always a single word, though preprocessing of the input creates some special words like *find_out* that are treated as tokens. See Section 4.4.3 on lexical processing and preprocessing. To recognize a token the parser identifies its part of speech, and related lexical information: is it singular or plural, a present or past participle, comparative, superlative, or possessive. Other information is also generated about classes of words that the token may be a member of: for example, pronouns and adjectives may be demonstrative or interrogative. These classifications are useful in particularizing the syntactic functions of high level constituents. Since many words act as several parts of speech, or belong to several word classes, the problem of ambiguity arises immediately. In keeping with the notion that parallelism is a natural solution in the problem of parsing, an

Sentence ~~> Main Clause

Clause :: Main Clause
Clause :: Conditional Clause
Clause :: Verbal Clause
Clause :: Interrogative Clause
Clause :: Relative Clause

Verbal Clause :: Infinitive Clause
Verbal Clause :: Present Participle Clause
Verbal Clause :: Past Participle Clause

Clause ~~> Verb Object, Noun Object, Adjectival Object, Adverbial Object

Object :: Verb Object
Object :: Noun Object
Object :: Adjectival Object
Object :: Adverbial Object

Noun Object ~~> Noun Phrase
Noun Object ~~> Relative Clause
Noun Object ~~> Interrogative Clause
Noun Object ~~> Infinitive Clause
Noun Object ~~> Present Participle Clause

Verb Object ~~> Verb Phrase

Adjectival Object ~~> Adjective Phrase
Adjectival Object ~~> Prepositional Phrase
Adjectival Object ~~> Appositive ~~> Noun Object

Adverbial Object ~~> Adverb Phrase
Adverbial Object ~~> Prepositional Phrase
Adverbial Object ~~> Conditional Clause

Phrase :: Noun Phrase
Phrase :: Verb Phrase
Phrase :: Adjective Phrase
Phrase :: Adverb Phrase
Phrase :: Prepositional Phrase

Fig. 4.1 Definitions of High Level Syntactic Constituents

input word produces as many tokens as it has alter egos. In this parser, lexical lookup also generates semantic information about the input word, but these pieces of information are not

carried with the token itself, but activated as part of the knowledge data base.

Above the *token* level is the *phrase* level. These are units tightly bound by syntactic order. They include noun phrases, prepositional phrases, adjective or adverb phrases, and verb phrases. To recognize *phrases* the parser follows the same sequence of steps that an ATN grammar would execute. It understands that there are a limited number of choices for the continuation of a partially recognized *phrase*. The definition of prepositional phrase is one of the simplest of the five: a prepositional phrase consists of a preposition and a noun phrase.[5] When the parser has seen a preposition, the next token will cause it to save the state of the partially recognized prepositional phrase, and initiate, if it can, recognition of a noun phrase beginning with the new token.

An adverb phrase consists of adverbs, e.g., *very quickly*; an adjective phrase of adjectives and possibly adverbs, e.g., *truly yours*. Adverb and adjective phrases exist only when they follow or are isolated from the objects they modify, like a predicate adjective. Otherwise adjectives and adverbs are embedded in noun phrases and verb phrases. Verb phrases consist of contiguous adverbs and verbs, with the constraint that the accumulation of verbs must follow the rules of tense formation. Noun phrases are the only really complicated *phrases*. They may be nouns, modified nouns, compound nouns with or without modification, or pronouns. They may be elliptical, such as the phrase *the good*. They may in fact be almost any word when it is used as a symbol, e.g. *never mind the why and wherefore*.

Keeping the definitions of *phrases* as simple as possible reduces the problems in recognizing them. This shifts the burden in parsing from a level that is primarily syntactic in nature to a higher level where semantic elements can exert greater influence. Apart from determining the allowed continuations in the *phrase* definitions, the other remaining problem in recognizing them is determining when they end. The parser must not fail to recognize an

[5]The theoretically best choice here would be *noun object* rather than noun phrase. This change has not been effected in the grammar, and is one of those belated discoveries that can be seen in hindsight. A separate definition of prepositional phrase is required at present for "preposition followed by a present participle clause." The parser needs this other definition to parse a question like *Can I end my mail message by hitting the break key?* This inconsistency allows the parsing of verbal clauses to be more consistent and easier, since it is the more complex problem.

elliptical noun phrase because it could not find a noun. It also should not try to join two adjacent nouns as a compound noun if they are two distinct objects. It has turned out that the definitions of verb phrase, adverb phrase, and adjective phrase can be restricted so that they must end with as verb, adverb, and adjective, respectively. The greatest difficulty lies with the noun phrase. The solution applied is to recognize in parallel as many noun phrases as there are appropriate boundaries for ending it. The incorrect versions will be discovered by semantic means or an eventual syntactic dead end.

A statement or clause (which is regarded as an embedded statement) consists of constituents of the second highest level, *objects*. They represent categories with overlapping boundaries, but their function in a statement can be determined clearly. A constituent at this level is either a *noun object* or a *verb object*, an object that modifies a noun - an *adjectival object*, or an object that modifies a verb - an *adverbial object*. These four classifications are reminiscent of the high level lexical categories applied by Cercone to individual words [Cerc 77]. He describes four "open" categories of lexicon entries that generally correspond to the parts of speech: noun, verb, adjective, and adverb. Words in "open" categories supply the content in a sentence, words in "closed" categories, such as prepositions, supply functional kinds of meaning.[6] However, Cercone does not apply these ideas to larger syntactic constituents. The grammar developed for UGURU takes as a premise that this kind of high level recognition of syntactic *objects* is an essential part of the way people understand language, and, therefore, should play its part in the analysis process.

The addition of the *objects* level offers a compromise between abandoning strong syntactic rules and being constrained by them. Syntax does not organize constituents at this level as tightly as it does at the lower levels, but supplies what might be called *principles* of structure that guide the order of wording in a statement. Together with semantic knowledge derived from the words and their references, the principles concern themselves more with the

[6] A category is "closed" when all possible members are asserted in the lexicon. Other "closed" categories include conjunctions, pronouns, and determiners.

presence or absence of essential and supplementary constituents than with a required order.

As a demonstration of how a statement may be organized at the *objects* level, we will consider a sentence that contains one of each of the types of objects. Its noun object is *the children*; its verb object is *raced*; its adjectival object is *shouting and jumping*; and its adverbial object is *into the playground.* The labelings N, V, N′, and V′ will be used to identify each constituent, respectively. Altogether there are twenty-four permutations of the order in which N, V, N′, and V′ can be rearranged. Without engaging in questions of literary taste, a full dozen of these permutations, representing perfectly acceptable English, are listed in Figure 4.2, though some do have a "dramatic" quality. In certain contexts, such as telling a story, a dramatic affect for highlighting may be the most appropriate arrangement.

Figure 4.3 lists another five permutations are listed that might best be characterized as "awkward." On reading these sentences, one is struck by their awkwardness, and perhaps feels some instinctive compulsion to rephrase them since the intended meaning is still clearly drawn. The separation of these "awkward" versions from those in Figure 4.4, the "wrong" versions, is not scientific, of course, but meant to emphasize that are degrees of unacceptability in the arrangements of words. Rules that govern our thinking on word order must, therefore, rely on certain kinds of thresholds. Even within the acceptable permutations in Figure 4.2, there are gradations of acceptability; some are more "natural" than others. These seem to be closely related to the principles that govern the inclusion of commas for clarity.

The principles that govern the presence and relative order of high level constituents are the principles that should drive the parser at the *object* level. Some of these principles are easy to see from Figures 4.2-4.4. The subject and verb must not be separated by too many other constituents, although a single constituent may be lengthy in itself, such as a modifying clause. The permutations where N and V are first and last, or vice versa, all appear among the worst choices. Permutations where V is first seem awkward at best: a special context, perhaps poetry, might make them credible. The permutations did not consider the reordering that takes place with questions, where the verb precedes the subject. However, since a help-

N : *the children*
N′: *shouting and jumping*
V : *raced*
V′: *into the playground*

NN′VV′	The children, shouting and jumping, raced into the playground.
NVV′N′	The children raced into the playground, shouting and jumping.
NVN′V′	The children raced shouting and jumping into the playground.
N′NVV′	Shouting and jumping, the children raced into the playground.
N′VNV′	Shouting and jumping, raced the children into the playground.
N′V′NV	Shouting and jumping, into the playground the children raced.
N′V′VN	Shouting and jumping, into the playground raced the children.
V′VNN′	Into the playground raced the children, shouting and jumping.
V′NVN′	Into the playground the children raced, shouting and jumping.
V′N′VN	Into the playground, shouting and jumping, raced the children.
V′N′NV	Into the playground, shouting and jumping, the children raced.
V′VN′N	Into the playground raced [the] shouting and jumping children.

Fig. 4.2 Acceptable Permutations of N, V, N′, and V′

NV′VN′	The children, into the playground, raced shouting and jumping.
N′NV′V	Shouting and jumping, the children into the playground raced.
V′NN′V	Into the playground, the children, shouting and jumping, raced.
VNV′N′	Raced the children into the playground, shouting and jumping.
VNN′V′	Raced the children, shouting and jumping, into the playground.

Fig. 4.3 Awkward Permutations of N, V, N′, and V′

N′VV′N	Shouting and jumping, raced into the playground the children.
NV′N′V	The children, into the playground, shouting and jumping, raced.
NN′V′V	The children, shouting and jumping, into the playground raced.
VN′V′N	Raced, shouting and jumping, into the playground the children.
VN′NV′	Raced, shouting and jumping, the children into the playground.
VV′NN′	Raced into the playground, the children, shouting and jumping.
VV′N′N	Raced into the playground, shouting and jumping, the children.

Fig. 4.4 Unacceptable Permutations of N, V, N′, and V′

ing verb is almost always used, and is separated from the core verb by the subject, the principles already mentioned still apply.

The shifting of modifiers, that is *adjectival objects* and *adverbial objects*, is remarkably fluid, and the restraints on such rearrangements operate with subtlety. The limitations derive in many cases from the "deep case" or pseudocase role that the modifying object plays. Adjectival objects in general have less pliancy than adverbial objects; certain forms of adjectival objects like relative clauses must follow the modified noun object. The respective order

of N and V seems less important than their proximity. The order where the subject precedes the verb is of course more frequent in normal prose and speech, but this may best be treated simply as a matter of frequency rather than correctness or incorrectness. The addition of a second noun object constrains this freedom a great deal however.

The proximity required between a verb and its object seems to be even stronger than between subject (**auth**) and verb. To demonstrate the constraints built up by the addition of multiple noun objects and modifying objects, consider a second example. The subject, N_1, is *the boy*; the verb object, V, is *hit*; the object, N_2, of *hit* is *the nail*; one adverbial object, V_1, is the adverb, *furiously*; the second adverbial object, V_2, is *with the hammer*. For the moment, no permutation includes the rewriting of any object, such as changing the verb to its passive form.

Out of 120 permutations of the five objects, only seven are acceptable sentences (see Figure 4.5.). The most severe constraint is the presence of the second noun object. The relative order of the noun objects and the verb object is limited to N_1VN_2. Generally the placement of the modifiers is flexible with the N_1VN_2 framework, but there are restrictions. One is that V and N_2 may not be separated; the proximity of the verb and its object override other arrangements. Also V_2, *with the hammer*, may not follow N_1, *the boy*. Because of the ambiguity of the word *with*, the arrangement N_1V_2 takes on a meaning different from the original. The preferred position of prepositional phrases is immediately following the modified object; if *with the hammer* appears immediately after *the boy* but before the appearance of *hit*, it will be identified as an adjectival object, not an adverbial object, with the sense of distinguishing a particular boy. The inference that the hammer is probably used for hitting will emerge later. When *with the hammer* appears at the end of the sentence, there are several possibilities for recognizing and binding it. The semantics of the situation must be used to assert the proper binding.

If certain changes are allowed to the *objects* in this example, the number of acceptable permutations grows appreciably. For example, if the adverbial object, *at the nail*, replaces the

N_1: *the boy*
N_2: *the nail*
V : *hit*
V_1: *furiously*
V_2: *with the hammer*

$N_1VN_2V_1V_2$	The boy hit the nail furiously with the hammer.
$N_1VN_2V_2V_1$	The boy hit the nail with the hammer furiously.
$N_1V_1VN_2V_2$	The boy furiously hit the nail with the hammer.
$V_1N_1VN_2V_2$	Furiously the boy hit the nail with the hammer.
$V_1V_2N_1VN_2$	Furiously, with the hammer, the boy hit the nail.
$V_2N_1VN_2V_1$	With the hammer, the boy hit the nail furiously.
$V_2N_1V_1VN_2$	With the hammer, the boy furiously hit the nail.

Fig. 4.5 Acceptable Permutations of N_1, V, N_2, V_1, and V_2

noun object, *the nail*, so that *the boy* is the single noun object, there are now fifteen acceptable orders. The placement of *at the nail*, which stands in the same sort of object relationship to the verb as *the nail*, is less constrained in its proximity to the verb. Although the object has been transformed in this way, the subject and verb still behave much as they do in the earlier example (Figure 4.2).

As an example of another transformation of an *object*, *hit* may be replaced with *was hit*; then *the boy* must be replaced with *by the boy*. Again, there is a single noun object remaining, *the nail*. Since the separation of the auxiliary verb from the main verb is normal, it is not considered a factor in the permutations here. Approximately twenty acceptable orders can be produced among these objects when the passive voice is used.

In sentences that contain clauses, the flexibility with which the high level constituents can be ordered still holds true. A subordinate clause is simply a noun object, adjectival object, or adverbial object from the viewpoint of the main clause, and may be reordered within the main clause like other objects in those categories. The constituents belonging to a subordinate clause may be reordered within the boundaries of the clause with much the same flexibility, although it seems to be constrained with rare exceptions to a subject-verb order, following the verb with simple *adverbial objects*. It is significant that there is a left and right boundary on clauses. A clause is contiguous with respect to its components: no *object* belonging to a subordinate clause is ever placed so that an *object* of the main clause separates it from the other *objects* of the subordinate clause. By defining the existence of *clause* structures and *object* structures, the parser must also be able to delimit these structures. The fact that the boundaries exist make this possible and occasionally advantageous.

The grammar defines five kinds of *clauses*: main, verbal, conditional, relative, and interrogative. The four named last are all subordinate clauses, and cannot constitute a sentence in themselves. All clauses but the verbal clause contain complete verbs, that is, a conjugated form with identifiable tense, person, and number. Verbal clauses are built on a present or past participle or infinitive. Conditional clauses may begin with *if*, *when*, *while*, *since*, or *because*, and act as *adverbial objects* in the parent clause, while relative clauses act as *adjectival objects* modifying *noun objects* and may begin with *that*, *which*, *who*, or perhaps missing the introductory word altogether. Interrogative clauses may act as either *noun objects* or *adverbial objects* and may begin with an interrogative pronoun, adjective, or adverb.

4.3.2.2. Recognizing Syntactic Constituents

The parser recognizes constituents through the principles of ordering, proximity, presence, and other syntactic and semantic means elicited from the study of examples like those in the last section. These principles are distinct from the transformations of a transformational generative grammar [Chom 57] and from the rules of a semantic grammar. The attempt to form a complete set of allowed transformations, were it even possible, would produce a set

of rules too complex and too large to be practical. Semantic grammars lean too far the other way by hardwiring the allowed permutations between given words or types of words. Semantic grammars are inherently myopic about the general principles of sentence structure.

The grammar used in UGURU attempts to treat the principles of language structure as forces that operate in parallel with one another, although with varying binding strengths. The implementation seeks to avoid the situation where the use of a rule depends on invocation by another rule, so that it becomes in essence a subroutine of the prior rule. It is not claimed that all, or even a majority of these principles are implemented; hopefully the design will allow the knowledge of other principles to be added as they are discovered.

The principles for recognizing the constituents in an input vary according to the level of the constituent. The parser processes lower level constituents primarily with the same step-by-step sequencing found in an ATN grammar. At the higher levels, it accumulates constituents according to several kinds of principles. Some of the principles concern the ordering of *objects* discussed above. Principles of proximity are some of the most important. Others require the presence of certain types of *objects*, and some disallow the addition of an *object* if certain others are already present.

Each constituent must have a syntactic identity, and every lower level constituent must be a component of one of the four types of *objects*. The parser recognizes any constituent in a top-down manner: for each new input word not belonging to the preceding *object*, it will generate guesses to cover all possible uses of the word, beginning with all *objects* that the word can initiate. It then elaborates each guess at the next lower level, maintaining a stack to trace the path followed. If a lower level constituent is a clause, the process may continue recursively. The logic of the program treats the multiple guesses as separate units whose processing occurs in parallel with one another. In Prolog, of course, the parallelism is simulated, but on the right computer, elaboration of the various guesses could indeed proceed in parallel.

The most essential element of any clause is its core verb. The principle of the core verb requires the presence of one, and only one, *verb object* per clause. Where two verbs are connected by a conjunction, the parser represents these as conjunctive verbs in a single clause only if all bindings are in common; otherwise it produces a *clause* representation for each verb, and duplicates the bindings that are common to both verbs. The question

Can I compile and link an assembly program in one step?

is an example of conjunctive verbs in a single clause. The question

How can I send some mail and find out if it got there?

consists of two clauses, whose common subject, *I*, must be bound to each of the two core verbs. Apart from conjunctive binding, a second verb sometimes occurs as a substitute *noun object*: in general, this means the second verb is a present participle or an infinitive. However, the parser considers such verbal forms to be the core verbs of clauses substituting in this case as nouns. The questions that follow all contain a main clause and a subordinate clause acting as a noun.

What is *compiling*?

How can *printing a file* be canceled?

How do I get *my program to quit* without the break key?

Of course, the existence of too many verbs may also indicate that a parse is not possible. For the present, the parser rejects an input that does not conform to the principle of the single core verb.

The presence of a *noun object* is also required in any clause, with the exception of verbal clauses acting as nouns or as *adjectival objects* modifying nouns. We commonly call this the subject, but this classification is of no use to the parser, and it must establish the "deep

case" instead. This may be any of **auth**, **obj**, or **indobj**, and may be unstated, that is, elliptical. Imperative statements are elliptical in this way. An example of an elliptical **auth** binding from the Unix domain is the following question in which the subject of the main clause is also bound as the subject of the conditional clause.

When working in the Bourne shell, can I still use the "!"?

An example of a verbal clause acting as a noun without the presence of a *noun object* within the clause is the question already quoted above, *What is compiling?*. Verbal clauses acting as *adjectival objects* will not contain their subject within their boundaries since the subject is a *noun object* belonging to the parent clause. The parser binds the modified object to the core verb of the verbal clause as well as attaching it to the parent clause. In the example that follows, *program* is bound as the **obj** of *interrupt* and the **auth** of *run[ning]*.

How do I interrupt a program running in the background?

There are several principles regarding the presence and absence of *noun objects* fulfilling the "deep case" roles of **auth**, **obj**, **indobj** and **purp**. A *clause* may contain at most one of each of these bindings, unless the several *noun objects* with the same binding are also connected by conjunctions. A *clause* must contain either an **auth** or an **obj** binding; neither an **indobj** or a **purp** binding may stand alone. The presence of an **indobj** binding specifically requires the presence of an **obj** binding. In general, only a single *noun object* with one of these bindings may precede the core verb, and only one follow. There are of course exceptions.

The most common exception is the sequence **indobj-obj**. This may occur immediately after verbs from certain verb classes, such as verbs of *giving*, *sending*, *making*, or *doing*.

How can I give my friend a copy of my file?

Indobj and **obj** bindings also may appear separated by a core verb that is a passive form of

one of the verbs above.

How can I be given a signal by the computer after an hour is up?

Here, *computer* fulfills the role of the **auth** of *give*, *signal* is the **obj**, and *I* is the **indobj**. Note that producing the correct bindings cannot depend on knowledge of the declensional forms of pronouns, i.e., that *me* is the correct form of the first person to use as an indirect object. That *I* is the indirect object of the question above is clear from comparing it to the following question, which is identical in meaning.

What command tells the computer to give me a signal after an hour is up?

Expressing the "indirect object" relationship as a prepositional phrase rather than a *noun object* relaxes the constraints on the order of the constituents in the sentence since there is one fewer *noun object*. The placement of the prepositional phrase has the relative freedom of an *adverbial object*, as we saw in the example illustrated in Figure 4.5. The correspondence of greater freedom and the use of prepositions has analogies with languages such as Latin that assign declensional endings to show the interrelationships of the words. If prepositions can be considered a substitute for declensions, then *noun objects* are elliptical declensions.[7] The constraints on the number and placement of *noun objects* helps to provide the missing information. The grammar developed for UGURU differentiates *noun objects* from other *objects* even though the underlying relationship may be the same, since that seems necessary to understanding the principles of syntax.

Another exception to the principle of a single *noun object* preceding and following the core verb is the placement of *noun objects* bound as the **purp** of the core verb. These *noun objects* are always *clauses*, and generally infinitive clauses. Their placement is as flexible as

[7]Though it was arrived at by a different avenue and different terminology, this statement expresses the same viewpoint as Fillmore's description of the *Kasus* symbol that identifies case. A key factor in creating and generalizing the symbol was that it was allowed to be null.

that of *adverbial objects*, and in fact an equivalent rephrasing of

What command do I use to bring a job back into the foreground?

is

What command do I use for bringing a job back into the foreground?

The purpose of *use* is expressed as a prepositional phrase in the second version. The verbal clause *bringing a job back into the foreground* is the *noun object* belonging to *for*. The syntactic distinction between the two versions is maintained during parsing because infinitive clauses act in general behave as *noun objects*. The parser binds *bring* as the **purp** of *use* in both cases. As far as placement is concerned, the following two alternatives of the version with the infinitive clause seem reasonable.

To bring a job back into the foreground, what command do I use?

What command to bring a job back into the foreground do I use?

The only restriction seems to be that it may not separate a core verb from its closest *noun object*, whether that is an **auth** or an **obj** binding.

Some rare exceptions will be noted here also. They are important because of what they tell us about the principles of English. While they are hardly ever used, particularly in conversation, they are still perfectly understandable. The principles that allow us to understand the vast majority of statements also allow us to understand these unusual statements. Consider the following:

In that multitude, found he many who believed.

The sweater I bought in Iceland, and the hat in England.

When the *noun objects* and *verb objects* are extracted, the first sentence contains the pattern **verb-auth-obj**, and the second the pattern **obj-auth-verb**. In both cases, the *noun objects* have accumulated on one side of the verb. For whatever reasons, the orders **verb-obj-auth** and **auth-obj-verb** do not ever appear, and this prevents some minor cases of ambiguity. The second pattern is actually fairly common in the form of questions:

Which escape sequences do I use to change the screen colors?

The only difference is that the auxiliary verb is detached from the core verb and separates the **obj** and **auth**.

The parser implements these principles by noting the presence of *noun objects* and *verb objects*. It allows nearly all possibilities that are legal, but assigns a greater weight to those that are most commonly used. The combination of these weights and further sifting by semantic checking enables the parser to select the right set of bindings. It disallows illegal possibilities, such as two **auth** bindings. When two *noun objects* follow each other consecutively, the situation may be one the unusual cases described above. There are other, more likely possibilities that the parser elaborates also. The two *noun objects* may be bound to one another, rather than both to the same core verb: the second, or even the first, may serve as an appositive to the other; they may together form a compound noun; a third possibility is that the second begins an elliptical relative clause modifying the first. For example,

How can I mail a file I have already created?

With the second occurrence of *I*, the parser has already bound two *noun objects*. The first *I* is the **auth** of *mail*, and *file* the **obj**. The possibility that *file* is the **indobj** will have been discarded by semantic checking. The parser therefore assumes a missing *that* between *file* and *I*, and begins recognition of a relative clause.

There is much less that can be said about the principles for binding *adverbial objects* and *adjectival objects* than those for binding *noun objects* and *verb objects*. Multiple bindings

of the same type are possible, though if they occur, it is generally implemented with conjunctions. A second occurrence of the same binding may therefore be assigned a very low weight. In terms of placement, modifiers usually follow right after the modified object. Proximity is awarded a high weight. Prepositional phrases cannot precede *noun objects* they modify; a prepositional phrase beginning a sentence is assumed to be an *adverbial object.* Some principles can be formulated that express a preferential order. For example, a gerund that follows a *noun object* that it modifies will in most cases have a trailing modifier of its own. This kind of principle helps to define the right boundary of the gerundive phrase.

The principles governing the placement of modifier *objects* are implemented in part through the syntactic checking of the parser, but even more by the semantic checking. It is the job of the syntactic component to make sure that the correct possibility exists among all those being considered. Finally, however, the only sure check on the binding derives from the semantic knowledge that the parser has.

CHAPTER V

THE PARSER: IMPLEMENTATION

The action of the parser is the sum of the individual actions of units called **recognizers**. Each recognizer contains some items of information pertaining to the input. It also has an implicit range of expectations, since this information will only match certain patterns found in its environment. The environment consists of input words plus lexical information about the words, other recognizers, and a set of pattern-action rules.

Each new input word begins a new cycle of activity. During the cycle the recognizers match as many patterns as they can. Logically they operate in parallel with one another; the relative order in which the recognizers match patterns during the cycle does not matter. Various conditions described below will cause the spawning of new recognizers or their destruction. The total number in existence at any one point grows from a single unit at the beginning of the parsing to possibly hundreds, increasing or decreasing with each cycle.

A cycle contains four steps, called *shift*, *reduce*, *match*, and *choose*. The first two steps implement syntactic updating of the recognizers; the *match* step interprets semantic information pertinent to the recognizers; the last step deactivates the less promising recognizers, so that only the strongest will continue to be expanded.

5.1. Nature of the Recognizers

The recognizers are divided into two classes: syntactic recognizers and semantic recognizers. A syntactic recognizer collects a set of bindings that represent a parse of the input. Each syntactic recognizer represents a different analysis of the input, so their number grows as the number of alternative parses grows. A semantic recognizer contains information about a specific constituent that appears among the bindings and the accepted usages for that constituent. Thus a syntactic recognizer carries a history of the parse to date, while a semantic recognizer knows only about one aspect of it.

All recognizers are declarative pieces of knowledge; that is, they are asserted into the data base. In Prolog, assertions and deassertions can be made in a straightforward manner. Descriptions of the implementation of parallel algorithms through assertions in the data base and the use of declarative knowledge in pattern-directed programming may be found in Chapter 16 in Bratko's *Prolog Programming for Artificial Intelligence* [Brat 86]. These ideas were a major influence in the design of UGURU's parser. Recognizers are independent of one another, with the exception that during the *match* step of a cycle, syntactic recognizers interact with semantic recognizers.

The primary inspiration for creating the parser from independent units called recognizers was the TLU model of neural networks [Rume 86], [Brow 87] (see Chapter II, Sec. 2.3.3). UGURU's current implementation has borrowed the framework more than the specifics. Because of the still considerable experimentation needed to create and train TLU networks to perform even small tasks, it would be unrealistic to hope to implement a program the size of a parser with them at this time. The themes that have been borrowed from the TLU

models are:

(1) the system is built from independent units where each unit has the task of recognizing a different component or aspect of a component of the larger problem presented to the whole set of units;

(2) inclusion of a weighting scheme to show relative excitation or inhibition of a particular unit;

(3) and the capacity to pass the output of one unit as the input of another.

In fact, recognizers behave very much like independent parallel processes that allow message passing. This was an unintentional direction that the development of the system took, but it was not unwelcome since the problems of handling parallel processes are more familiar and better understood than TLU networks.

5.1.1. Syntactic Recognizers

Each type of recognizer contains five arguments. Those belonging to a syntactic recognizer are listed in Figure 5.1. The first two arguments carry information only about the constituent currently being recognized, whereas the last three carry information that reflect the entire history of the recognizer. Each syntactic recognizer has the goal of recognizing a single

---

(1) state;
(2) current stack of bindings;
(3) cycle number;
(4) weight;
(5) genealogy.

Fig. 5.1. The Arguments of a Syntactic Recognizer

---

constituent that will either be a *clause*, *object*, or *phrase*. In the course of identifying a higher level constituent, a recognizer may be suspended while another is spawned to recognize one of its components. The child that is spawned stores the knowledge of the parent so that it may be reasserted when the child's work is done. A parent will spawn a child for each alternative continuation of the parse that is possible.

The state of the recognizer identifies the level and kind of constituent, plus an indication of the components that have been found so far. The notation for writing the state uses the operator // to separate the constituent and its components. For example,

phr(noun)//adj(s)

indicates that the recognizer is looking at a noun phrase, and that the last component seen is an adjective (simple). Since the recognition of a *phrase* constituent proceeds logically like a simple ATN or even a finite state machine, **adj(s)** is enough to pinpoint the state within the noun phrase.

A possible state attained during recognition of a relative clause might appear like this:

cls(rel^that)//[auth] ^vb(cvb^actv).

A recognizer in this state is currently absorbing a relative clause introduced by *that*, for which a subject (presumably *that* and indirectly its referent) and a core verb have been found. The components of the clause following the // separator are divided by the caret operator: on the left is a list of the bindings from *noun objects* to the core verb, and on the right is information about the core verb. In this example the subject is the **auth** of the verb, and the verb has taken its active form. The presence of *noun objects* and *verb objects* is marked as part of the state; the state does not reflect *adverbial objects* and *adjectival objects* since these constituents do not determine whether the clause is structurally complete in a syntactic sense, even though they may be required to complete the semantic sense. This follows directly from the principles of *clausal* organization discussed in Section 4.3.2. The *noun objects*, since there may be more than one, are maintained in a stack so that the order of appearance can be used when a situation requires it. Actually this seldomly occurs, generally when both an indirect object (without a preposition) and an object follow the verb. The

indirect object must precede the object (see Section 4.3.2.2). If an object were subsequently found as a continuation of the relative clause in this example, the parser would update the state as follows:

cls(rel^that)//[obj,auth] ^vb(cvb^actv)

The :: productions in Figure 4.1 show the different constituent states that may be entered by a recognizer. On the left side of the // operator in the state description syntax, the allowed *phrase* states are written as **phr(noun)**, **phr(vb)**, **phr(adj)**, **phr(adv)**, and **phr(prep)**. The allowed states for *objects* and *clauses* are formed in a similar manner based on the *object* and *clause* :: productions. The *clause* states generally include other information that particularizes the clause by incorporating detail from its use in the given input. For example, a relative clause will include its introductory word, as above. For a main clause, the parser will also assign a *mode* when enough input has been consumed so that it can be determined. The mode is either **st** (statement), **qu** (question), or **imp** (imperative). For a sentence beginning with the word *when*, for example, the parser will inaugurate three alternative parses. One will assume that the continuation will reveal a question: *when* is contained in a main clause with the state

cls(main^qu)

A second will assume that a conditional clause has begun; the conditional clause must of course be an adverbial constituent of a main clause. The state of the conditional clause is known:

cls(cond^when)

but the main clause could eventually become either a statement or a question. The parser withholds judgment on this point until it can be determined or a reasonable guess can be made. However, the parser does establish certain relevant information. The parent clause is a main clause whose mode will be represented by an uninstantiated variable. In Prolog, the state of the parent clause would appear

cls(main^_)

A sentence beginning with *when* also has a third alternative parse that is known to the system.

Here, *when* begins an interrogative clause acting as a *noun object*, e.g., *When* to buy a car depends on interest rates. The mode of the parent clause can be safely assumed to be st.

The current stack contains the bindings drawn from the current constituent. The bindings may be complete or incomplete as they are stored in the stack. Incomplete bindings will ordinarily be referred to as half-bindings in the text that follows.

When a recognizer is constructing a noun phrase, such as *a very fast computer*, it knows that any adjectives encountered will modify an eventual noun or noun substitute, and the adjective and the binding relationship **adjmod(s)** can also be stored. When parsing has processed the first three words in this example, *a very fast*, at least one syntactic recognizer will active in the state

phr(noun)//adj(s)

The current stack belonging to such a recognizer will contain two half-bindings: *a* and *fast* are both known to have an adjmod(s) relationship. Whenever the noun is shifted into the recognizer, the half-bindings can be completed. In the case that the noun phrase is elliptical and a noun does not appear, the referent must be known from the context. Since the parse is not complete until all individual bindings are complete, the parser will be forced at some point to locate the object being modified.

In the same example, again just prior to reading the noun, a recognizer absorbing the noun phrase will also contain a third binding in the current stack. When the adverb *very* was processed, it was initially stored as a half-binding with the binding relationship **advmod(s)**. For an adverb in a noun phrase, the next word shifted in, as long as it is an adverb or adjective, is the modified word. In this example, when the recognizer sees *fast*, it will complete the binding between *very* and *fast*, and initiate the **adjmod(s)** binding between *fast* and the unknown but expected noun. Therefore, after seeing the first three words, the current stack will contain two half-bindings built from the adjectives, and a full binding between *very* and *fast*.

The modifiers contained within a noun phrase always precede the modified object. *Adverbial objects* and *adjectival objects*, since they are recognized within the *clause* level rather than the *phrase* level, may either precede or follow the objects they modify. Completing bindings can be handled in the same way at both levels if the modifier precedes, but at the higher level there are more possibilities to consider since the order of *objects* is quite flexible. Modifiers may not only precede or follow, but may be separated by intervening objects from the modified object.

If a prepositional phrase occurs at the beginning of a sentence, it must be stored in the stack belonging to the clause with an incomplete binding. Since the grammar does not allow prepositional phrases that modify *noun objects* to precede them, a prepositional phrase found at a sentence's beginning must modify a verb.[1] When the core verb has been recognized, the binding of the prepositional phrase can be completed. The following statement contains such an introductory prepositional phrase.

In one hour, the whole network will be shut down.

Recognizers operating at the *clause* level will see this statement as an *adverbial object*, followed by a *noun object*, a *verb object*, and another *adverbial object*. For its state, some of these recognizers will assume from the outset that this is a main clause, and after recognizing the *noun object* and its location, that it is a statement. Recognizers are spawned that identify the lower level *objects* and *phrases*. When *In one hour* has been identified as an *adverbial object*, a *clause* recognizer will take the bindings produced by the child recognizers and include them in its own current stack. In this case, there is a subsidiary **adjmod(s)** binding between *one* and *hour*. The binding that allows the prepositional phrase to be attached to the sentence as a whole is, at the moment, a half-binding. The system records it as an

[1]Special semantic mechanisms exist to cover the rare exceptions to this principle. The mechanisms allow the recognition of prepositional phrases beginning with *like* or *of* to precede the modified *noun object*, e.g., *Like his father in many ways, Joe has grown up quickly.* Construing *like* as an adjective would prevent having to treat this as a special case, but, given its common usage as a preposition, it would be better to avoid artificial solutions.

advmod(_^in) binding from *hour*. A semantic recognizer with the appropriate semantic knowledge will be able to more completely specify the half-binding as **advmod(when^in)**.

The half-binding from *hour* will remain so during the time that the *noun object* and *verb object* are recognized. When the recognizer at the *clause* level receives the verb, it will replace the half-binding from *hour* with a full binding that connects it to *shut*.

On the other hand, a prepositional phrase located near the end of a *clause* can potentially modify any *noun object* or the core verb if only syntactic order is considered. Multiple recognizers must be spawned so that each possibility may continue to be traced. The parser has access to semantic knowledge regarding the accepted combinations of prepositions and classes of verbs and nouns and therefore can eliminate some of these possibilities directly. Further semantic or contextual knowledge must be applied to sort out the other cases: the parser can achieve this by adding weight to preferred combinations, and reducing the weight on unfamiliar or unlikely combinations. As an example, consider the request

Please give me the name for the directory with system binary files.

The proper attachment of the prepositional phrase *for the directory* and *with system binary files* involve important semantic issues in parsing the statement. UGURU's parser will use the proximity of *for* to *name* and *with* to *directory* to add some weight to these attachments. *Give* regularly takes prepositional phrases beginning with *for* and *with*, however, and these occasions must also be handled properly.

I gave him the book for cash.

I gave him the book with pleasure.

The solution is to allow semantic knowledge to surface during the parse that adds higher weight for the strong combination of *name* and *for*, and *with* to any noun if the prepositional object is a component of that noun.

The completion of the **auth**, **obj**, and **indobj** bindings also may require two separate transactions. When a *noun object* precedes the core verb, it must necessarily be stored in the

clause's current stack as a half-binding that can be completed when the core verb is found. If the *noun object* occurs in the predicate, the complete binding may be created at once. Whether the verb is yet present or not can be deduced directly by the system from the state argument within a *clause* level recognizer: absence of the core verb is notated as a 0, e.g.

cls(main^st)//[auth]^0

The system always initializes a *clause* recognizer to include the *0* value.

The genealogy of a syntactic recognizer, stored as its fifth argument, contains the information showing the history of the parse prior to beginning the recognition of the current constituent. This is in fact the representation of the parent recognizer from which the current one was spawned. The parent recognizer is necessarily incomplete, and once the current constituent is recognized, the current recognizer is absorbed by its parent. Absorbing the child reactivates the parent, who incorporates the child's set of bindings onto the top of its own current stack. The parent may itself be a child of some other syntactic recognizer, and when it has finished its assigned task, it will likewise be absorbed by its parent. Every recognizer must have a parent except those recognizing main clauses. A syntactic recognizer in the process of recognizing *I* in the question

If I send some mail, how do I know it got there?

shows this. The current recognizer would be trying to identify *I* as part of a *noun phrase*. A *noun phrase* is a special case of its parent, a *noun object*. The pronoun is being recognized in the midst of a conditional clause, and, therefore, the parent of the *noun object* is a *conditional clause*, which itself is an *adverbial modifier*. The recognizer that began processing the *conditional clause* was spawned by the one recognizing an *adverbial modifier*, which was spawned by the *main clause*.

Each step in a parsing cycle may involve any or all of the syntactic recognizers. All steps but the *match* step involve only the processing of syntactic recognizers, not the semantic recognizers. During the *shift* and *reduce* steps, each syntactic recognizer tries to assimilate

the current input word; if it succeeds, it will push a half-binding or a full-binding onto its current stack and update its state, and it may additionally complete previously stored half-bindings. It may be absorbed by its parent if it has finished recognition of its assigned constituent. During the *choose* step, the system sorts the syntactic recognizers and deactivates the ones that represent the least likely parses.

5.1.2. Semantic Recognizers

Like the syntactic recognizers, the semantic recognizers have five arguments. They are shown in Figure 5.2. Semantic recognizers represent activated semantic knowledge. They are created in response to the tokens produced by lexical processing and the other constituents and bindings produced by the syntactic recognizers. When a constituent appears, the system spawns a semantic recognizer for each item of semantic knowledge related to the constituent. Thus only knowledge that can potentially affect the parsing process is activated, and most of the semantic knowledge the system has will remain inactive. The activated knowledge contained in the semantic recognizers is used during the *match* step in a parsing cycle. In this step semantic recognizers are matched with syntactic recognizers about which

---

(1) the particular binding about which the recognizer has semantic knowledge;
(2) a representation of a syntactic recognizer containing the given binding (argument (1)), though generally this argument is an uninstantiated variable whose values will be taken from the syntactic recognizer matched through the given binding;
(3) a replacement syntactic recognizer that will either duplicate values from the syntactic recognizer matched (argument(2)), add to them, or replace them;
(4) a weight;
(5) further conditions for successfully matching a syntactic recognizer (argument (2)) in addition to the given binding (argument(1)).

Fig. 5.2 The Arguments of a Semantic Recognizer

---

they have a piece of pertinent semantic information. The semantic recognizers processed during the *match* step contain only a small fraction of all the semantic knowledge within the system. Activating only a relevant (or possibly relevant) portion gains efficiency, and is arguably a more psychologically realistic approach. This idea is used in several NLP programs, such as MARGIE [Scha 81], and the "spreading activation" technique in the Waltz and Pollock model [Walt 85].

In the lexicon, the entry for *contain* appears as follows:

```
vb(contain) :-

        semantcs( subvbs_need_objs, contain ),
        semantcs( vbs_no_indobjs, contain ),
        semantcs( vbs_and_preps, contain ),
        semantcs( vbs_and_auths, contain ).
```

By referencing this entry in a look-up of the word *contain* during the stage of lexical identification, the system automatically deploys four semantic recognizers. In this case, the first will inspect syntactic recognizers to insure that any clause with *contain* as its core verb must also have an **obj** binding between the verb and an object. The second semantic recognizer will disallow syntactic recognizers that have produced **indobj** bindings between *contain* and some object. The third and fourth do not implement criteria for eliminating syntactic recognizers altogether as the first two do, but effect preferences for certain recognizers by boosting their weight arguments (argument (4) in Figure 5.1). The third rewards syntactic recognizers that have bindings representing common or preferred uses of prepositions with *contain*, such as *in*. This helps to clarify the proper attachment of prepositional phrases. The fourth semantic recognizer rewards **auth** bindings that fall into classes of reasonable agents for the verb. This means, for example, that the system will treat a parse where a *file* does the containing as more credible than one where *characters* do. It is possible, of course, to make a meaningful statement, such as *These characters contain a secret message*, where *characters* is the legitimate **auth** of *contain*, but more usually, and in particular in the domain of Unix knowledge, *characters* would be the **obj** of *contain*. The weights enhance the preferred or more likely combina-

tions.

Much of the knowledge exemplified by these recognizers is based on the constraints of the words involved. A word may place constraints on the organization of the other words around it or govern the particular choice of words associated with it. A common example is the transitive or intransitive nature of verbs. The lexicon entry for *contain* includes a strong requirement for the presence of an object. Another common example is the association of certain prepositions and certain verbs: we might "communicate *with*" someone, but we would "write *to*" them. The third semantic recognizer generated from the lexicon entry for *contain* has access to a table of combinations[2] in which it can check for preferred combinations of its argument (in this case, *contain*) and prepositions that may be associated with it. The table is used when the semantic recognizer is processed during the *match* step.

It is arguable that these kinds of constraints are not really semantic knowledge as much as an extension of syntactic rules that apply only to individual symbols, since their use does not display any "understanding" of the meaning of the words. From the point of view of system design, they are important because they are able to decide a large percentage of ambiguous parses where several alternative sets of bindings exist. Frequently, one of the choices will have accumulated a higher weight than the others and so will be preferred. Sometimes clear cut decisions can be made about eliminating a choice altogether when a member binding is prohibited by the constraints. The first three semantic recognizers generated by a look-up of the word *contain* illustrate both these kinds of decision processes.

Semantic recognizers provide a single mechanism that can incorporate knowledge that is more genuinely semantic as well as the more word-specific knowledge of constraints. The fourth recognizer generated for *contain* inspects the **auth** binding to determine if the *noun object* bound has the ability to perform containing. Again, a table is used, although this table comprises names of classes as well as specific objects. For example, the table would allow

[2]The combination values are asserted as individual facts in the Prolog database. In essence, they constitute a table.

the combination of *file* and *contain*. Since a *program* can be a specific instance of a file,[3] the semantic recognizer also would allow the combination of *program* and *contain*. It would disallow a combination with *human*, or instances of human beings. Knowledge of general class types and instances of the classes is implemented through two relationships, **instof** (instance of) and **cmptof** (component of)[4] (see Section 4.3.1). These are also bindings that the parser may apply to input sentences. One of UGURU's strongest points may be the unity of implementation between parser output and stored knowledge.

Once spawned, a semantic recognizer normally remains active until the entire input has been parsed. The parser does not currently handle inputs of several *main clauses*. This is one situation where overlapping recognizers could cause confusion, although a solution could be provided easily: the act of recognizing a *main clause* generates a call to the system to deassert any or all semantic recognizers. Throughout a *main clause*, the semantic recognizers continue with each cycle to inspect the syntactic recognizers. This allows the knowledge they have to identify words or bindings that may be separated by intervening words or bindings. This scheme is particularly well-suited to handle idioms, especially the type that combines verbs and prepositions or particles. Among the semantic recognizers spawned for the verb *print* is one that looks for the presence of the word *out*. Assuming that *out* follows the verb, the recognizer will edit any syntactic recognizer that is beginning to build a prepositional phrase with *out* (likely but not guaranteed to fail), and make a preferred copy with *out* bound to *print* as an adverbial modifier. This will parse either *print out several files* or *print several files out*. An identical call in the lexicon entry for the word *throw* would allow the phrase *throw the baby and the bath out the window* to be parsed correctly, since the prepositional phrase *out the window* will continue to be parsed in at least one syntactic recognizer while those created by the semantic recognizer which treat *out* as an adverb will eventually fail to

[3]By common usage, e.g., *How can I compile my program?*

[4]These two relations correspond to IS-A and HAS-AS-PART, which are standard terminology in the literature on semantic nets [Schu 79]. Levesque and Mylopoulos describe a semantic network based on IS-A and PART-OF relations, and they feel that basing a network on the two basic "orthogonal" hierarchies is preferable to adding in greater complexity, and, along with it, greater "intransigence." [Leve 79].

be able to bind *the window*, since the **obj** binding already exists for *the baby and the bath.*

The semantic recognizers embody a pattern-action logic for the interactions with the syntactic recognizers. The pattern comprises the given binding (argument(1) in Figure 5.2), the conditions (argument (5)), and occasionally the description of the syntactic recognizer to be matched (argument (2)). If the entire pattern is matched, the action taken is to spawn a new syntactic recognizer with the bindings and weights adjusted to reflect the knowledge of the semantic recognizer. The matched syntactic recognizer may or may not continue to exist.

The matching process in turn pairs each semantic recognizer and each syntactic recognizer. It first attempts to equate the given binding (argument (1)) in the semantic recognizer with the top of the current stack (argument (2)) in the syntactic recognizer. The given binding may specify any, all, or none of the elements of a binding triple. When an element is not specified, it can be matched by anything. The given bindings must specify a full binding or a half-binding, however.

As a simple example, the code for a semantic recognizer is given in Figure 5.3. (Section 6.1 explains the syntax for writing bindings.) The recognizer's goal is to disallow the use of nominative forms of pronouns in object type bindings. The variable **Pron** is instantiated to a

```
semrcgnzr( Pron:_//_>---Rel--->_ ,          % argument (1)
           SynR,                            % argument (2)
           _,                               % argument (3)
           _,                               % argument (4)
           ( ( Rel = obj  ;  Rel = indobj ),
             retract( SynR ), fail ) ).     % argument (5)
```

Fig. 5.3 A Semantic Recognizer for Nominative Pronouns

specific nominative-only pronoun, like *I* or *he*, but excluding *it* for example. The given binding specifies that the variable **Rel** is the type of binding, and the modified object is left unspecified, written as an underscore. If a syntactic recognizer had formed a **obj** binding between *I* and *print*, for example, the two recognizers would pass the first test in the matching process. The last argument in Figure 5.3 specifies the additional conditions that must be made for a match. In this case, the variable **Rel** must have matched either an **obj** or **indobj** binding. The conditions are the second part of the matching test; if they are met, the match is a success, and the appropriate actions will be taken.

In general, the action taken is that a new syntactic recognizer, argument (3) in Figure 5.2, is created by duplicating the knowledge of the matched syntactic recognizer, argument (2) in Figure 5.2, and adding to it, or editing it in line with the specifications of the semantic recognizer. The weight of the old syntactic recognizer is adjusted and the new value assigned to the new syntactic recognizer. The weight may be adjusted up or down, to reflect the preference or disapproval, respectively, of the semantic recognizer. The example in Figure 5.3 shows a less frequent case. The conditions also include a call to **retract**, i.e., destroy, the old syntactic recognize, and by causing an artificial failure at this point, no new syntactic recognizer can be asserted. For this reason, an unspecified value (the underscore) is supplied for argument (3).

The variety of semantic recognizers that can be designed is essentially unlimited. Any level of knowledge - word constraints, word classes, contextual, or inferential - is possible. The semantic recognizers may be spawned by look-up in the lexicon during token identification, or they may be spawned by practically any other occurrence in the parsing if it seems appropriate. New semantic knowledge can be added to the system merely by adding another recognizer call to a lexicon entry. In a sense, UGURU's parser is a laboratory: since all syntactic avenues remain open, with the exception of those that eventually reach a dead end, parsing becomes a study in what semantic recognizers must be added to cause the proper choices to be made among the alternative syntactic paths at a psychologically realistic time.

5.2. Pattern-Action Rules

Besides the pattern-action directions embodied in the semantic recognizers, there is a set of rules that interact with the syntactic recognizers, also based on pattern-action logic. They are formulated as declarative pieces of knowledge, asserted into the data base. These rules are divided into two groups, *shift* rules and *reduce* rules, which correspond to the shift and reduce steps of a parsing cycle. The system currently holds somewhat less than 150 rules altogether, with approximately twice as many shift rules as reduce rules. In some ways, the system has been "prototyped" [Kell 86], as was intended, so that the system could begin with a small base of declaratively encoded rules, and be built incrementally.

The rules implement syntactic knowledge with few exceptions. During each parsing cycle, each rule is applied to each of the syntactic recognizers. If the pattern contained in the rule matches the recognizer, the rule supplies an action to be taken regarding the recognizer.

5.2.1. Shift Rules

A shift rule attempts to shift the input word onto a recognizer's constituent stack. Shifting normally means creating a half-binding from the input word, pushing the half-binding onto the constituent stack, and resetting the state of the recognizer. If the input word is shifted as the beginning of a new *object*, this will cause the current stack to be stored as part of the genealogy of the recognizer, and the new half-binding begins a new current stack.

Each shift rule whose pattern matches a syntactic recognizer will assert a new one with the adjustments made according to the action part of the rule. The original recognizer may be replaced by several new recognizers, each one representing a different continuation of the parse tree. If all the rules fail to match, the original recognizer will not be replaced, and so will disappear. This situation occurs when a syntactic dead end is reached. Syntactic recognizers die only through the inability to match shift rule patterns, or when deasserted by a semantic recognizer. In the question,

How can I send friends on VAXA mail?

friends will be shifted by a rule that marks it with an **obj** binding, and another that marks it with an **indobj** binding. There are no straightforward shift rules that are able to shift *mail* into the first of the new recognizers that were created, since only one **obj** binding is allowed (without conjunctions), and an **indobj** binding must precede an **obj** binding.[5] The recognizer with *friends* as **obj** will die since there are no continuations. The processing done by semantic recognizers in the cycle recognizing *friends* should also have boosted the weight of the recognizer with the **indobj** binding so that it was the preferred parse from that cycle in any case.

A simple example of a shift rule is one that accepts a noun into a noun phrase following an adjective. Figure 5.4 lists the actual code. The "pattern" the rule seeks to match includes:

(1) an input token that is a noun - represented by the variable **WdInfo**;

```
17#shiftTBL( WdInfo, RCGNZR, NewRcgnzr ) :-
      WdInfo = _:N//noun(_),
      RCGNZR = rcgnzr( phr(noun)//adj, Edges, _, Wt+SWt, Gen ),
      fixwt( 100, Wt, NewWt ),
      NewRcgnzr = rcgnzr( phr(noun)//noun,
                          [WdInfo>---nn|Edges],
                          N,
                          NewWt+SWt,
                          Gen ).
```

Fig. 5.4 Code for Shift Rule #17: "Accept a noun in a noun phrase where the previous word was an adjective."

---

[5]There are still some alternatives at this point nevertheless. For example, one alternative is a rule that will attempt to form a compound noun from *VAXA* and *mail*, a not unfeasible idea if the context of the rest of the sentence were different.

(2) a syntactic recognizer - represented by the variable **RCGNZR**.

The "action" taken is to create a replacement syntactic recognizer - represented by the variable **NewRcgnzr** - in which the state has been updated to reflect the addition of the noun, and a half-binding has been pushed on the top of the current stack that indicates that the input word is the noun, or part of the noun, that completes the noun phrase. The shift rule does not express in any way that a *noun phrase* has now been successfully recognized. It essentially states that a noun following an adjective in a noun phrase is a legal syntactic continuation. The reduce rules determine what the continuation is, if anything.

The "action" taken by the rule also includes calculating a weight for the new syntactic recognizer. It is a combination of the weight of the matched syntactic recognizer, **RCGNZR**, and a value specified by the rule that indicates the probability that this continuation belongs to the correct parse (see Section 5.3).

More complicated shift rules may cause the creation of a genealogy of several levels in the replacement recognizer. Rule #16 initiates conditional clauses when a conditional conjunction like *if* or *when* occurs in the input. The code is listed in Figure 5.5. The rule assumes that the recognizer to be matched is operating at the *clause* level. This is a natural

---

```
16#shiftTBL( WdInfo, RCGNZR, NewRcgnzr ) :-
      WdInfo = WD:N//conj(cond),
      RCGNZR = rcgnzr( cls(_)//_, _, _, Wt+SWt, _ ),
      fixwt( 95, Wt, NewWt ),
      NewRcgnzr = rcgnzr( cls(cond^WD)//[]^0,
                          [],
                          N,
                          NewWt+Swt,
                          rcgnzr(obj(adv)//0, [], _, _, RCGNZR ) ).
```

Fig. 5.5 Code for Shift Rule #16:
"Begin a conditional clause that will serve as an adverbial object in the parent clause."

---

assumption since conditional conjunctions will rarely occur as elements in *phrases*. Parsing is then almost certainly to be between *objects* within the *clause*, i.e., some *object* has just been recognized. The state of the old recognizer, **RCGNZR**, only specifies that it is at the *clause* level, and so it is allowing *adverbial objects*, such as a conditional clause, to occur anywhere among the objects in the clause.

The replacement recognizer in Rule #16 initializes the new state as a clause in which neither *noun objects* nor the core verb have been recognized. The genealogy specifies that, when the the conditional clause has been completely recognized, it will become an element of an *adverbial object*; the *adverbial object*, when completely recognized, will become an element of the current recognizer that has been matched by the shift rule. The rule accomplishes this return to the original by asserting **RCGNZR** to be the genealogy of the *adverbial object*.

It may be surprising that the conditional clause could be considered only an element of the *adverbial object* and not simply equal to it. The parser grammar allows any *object* to be "conjunctive," two or more of the same sort of object joined by connective conjunctions, *and*, *or*, or a comma with an eventual *and* or *or*. Thus, the adverbial object could be two conditional clauses joined by a conjunction. The state for the *object* indicates the presence of a conjunction by the value following the // separator. In rule #16, the *adverbial object*, as well as the conditional clause, is being initiated by the conditional conjunction, and so the state that will be assigned to the *object* level recognizer reflects this by using a 0 value:

obj(adv)//0

A number of shift rules have been considered that have not been added, for example, one to handle the rare case where a prepositional phrase precedes the noun it modifies. This new rule would create another alternative for the definition of *noun phrase*. To keep the search space small, however, it was decided that cases such as these should be handled by semantic recognizers.

5.2.2. Reduce Rules

Reductions performed by the reduce rules on syntactic recognizers do not replace the originals. If a reduction of a stack of bindings can be made, it will be, and a new syntactic recognizer formed by the reduction will be asserted. The original still exists, capable of shifting in additional words before a reduction is made. In other words, reductions are performed whenever possible and as soon as possible.

One of the simplest examples is reduction of a *verb phrase*. A verb phrase can only be recognized once the core verb is processed. This means that a shift rule will have pushed a core verb onto the top of the current stack belonging to some syntactic recognizer, updating its state to

phr(vb)//vb

On the stack, beneath the core verb, may be a combination of several adverbs and auxiliary verbs.[6] The stack may be empty, except for the core verb. When the syntactic recognizer with this state and current stack are matched against the patterns in the various reduce rules, it will activate Rule #509. The code for this rule is listed in Figure 5.6.

```
509#reducTBL(  Rcgnzr, NewOne ) :-
        Rcgnzr = rcgnzr( phr(vb)//vb, Edges, N, Wt+Swt, Gen ),
              Edges = [WdInfo >--- cvb | _ ],
              Gen = rcgnzr( obj(vb//State, CEdges, _, _, G ),
        lkEDG_vb( WdInfo, Edges, NewEdges ),
        append( NewEdges, CEdges, EdgesAll ),
        fixwt( 95, Wt, NewWt ),
        NewOne = rcgnzr( obj(vb)//State, EdgesAll, N, NewWt+SWt, G ).
```

Fig. 5.6 Code for Reduce Rule #509

[6] An auxiliary verb, like *is*, produces two tokens by the lexical processor, **getTOKinfo**. One token identifies it as an auxiliary verb, which cannot be reduced without accompanying a core verb. The second token identifies it as a core verb in its own right, which may then be reduced by Reduce Rule #509.

The first two lines of the code specify the required state and the top value on the current stack, variable **Edges**, belonging to the matched recognizer, variable **Rcgnzr**. This is the pattern in the rule that must be matched. The action the rule specifies includes:

(1) complete any half-bindings in the current stack. For example, any unattached adverbs appear on the stack in the form

some_adverb >--- advmod(s)

The subroutine **lkEDG_vb** adds the right arrow binding to the newly identified core verb. The half-binding is replaced with the form

some_adverb >--- advmod(s) ---> *core_verb*

lkEDG_vb completes all half-bindings. The only half-binding that may remain is that from the core verb itself. The attachment from the core verb can only be known from the *clause* level, since this verb may be contained within a verbal clause or other subordinate clause rather than the main clause. The type of *clause* environment in which this verb phrase is being recognized is contained in the genealogy stored in variable **Gen**.

(2) The current stack with completed bindings, **NewEdges**, is appended to the top of the parent stack, **CEdges**. Since the syntactic recognizer processing the *verb phrase* has completed its work, the values contained in its current stack can be absorbed by the parent recognizer, which can continue processing its assigned constituent that contains the *verb phrase* as an element.

(3) The weight associated with the probability of this reduction is factored into the accumulated weight of the current recognizer, **Rcgnzr**, and this adjusted weight will be passed back to the parent, **Gen**, when it is reactivated.

(4) The parent recognizer, **Gen**, that spawned the current recognizer, **Rcgnzr**, is recreated with updated values. Its previous state, cycle number, and genealogy are retained, but it has updated values for its current stack, **EdgesAll**, and weight, NewWt[+SWt].[7] The

[7]The variable **SWt** is a separate weight maintained and adjusted by the semantic recognizers alone. The first weight in the pair of weight variables is figured only by the actions of the shift and reduce rules.

recreated recognizer is returned to the system as the argument **NewOne**, and will be asserted into the data base of all recognizers.

The actions of the reduce rules effect the reassertion of the parent of the matched recognizer with updated values. They do not cause the deassertion of the matched recognizer, which will remain in existence at least into the next parsing cycle. The reduction performed on the current recognizer may be premature, as it turns out, when more of the input has been seen. If a reduce rule causes an unlikely or semantically impossible recognizer to be asserted, the system depends on the associated weights and the actions of the semantic recognizers to eliminate it. For *verb phrases*, this is not the case, and these comments do not apply to Reduce Rule #509, since the grammar of the system defines a *verb phrase* to contain one and only one core verb. There are no other possible reductions.

Reducing a *noun phrase*, on the other hand, is more complex, since UGURU's grammar provides that it may terminate at a variety of points. This is true of many constituents, particularly those, like clauses, that allow optional trailing modifiers or objects. A noun phrase in normal situations will be reduced once a noun (or pronoun) has been identified. However, reductions must be allowed for those cases where an adjective stands as a noun, or the noun phrase contains a compound noun. In the first case, a reduction must be permitted after the adjective. In the second, a recognizer that has found a single noun must continue to exist, as well as spawning a reduction, so that, if a second noun is found, it can be shifted into the recognizer and semantic recognizers must test to see if the combination of the two nouns is appropriate. A noun phrase that would cause three separate reductions in three separate parsing cycles is *the good computer system.* Those reductions that select *the good* and *the good computer* will most likely lead to other syntactic dead ends when a proper reduction for the isolated token *system* cannot be found. Since *the good* and *the good computer* are legitimate noun phrases, however, the system must produce syntactic recognizers by reduction that represent these possibilities.

Reductions may be possible in a recursive manner if the genealogy of the recognizer has several generations buried in it. Referring again to the example used earlier, *If I send some mail, how do I know it got there?*, several reductions will occur in the same parsing cycle that processes *mail.* The reductions will follow in reverse order the genealogy described above for the original spawning of child recognizers. Once *some mail* is reduced as a *noun phrase*, it can also be reduced as a *noun object* used as an **obj**. The *conditional clause* containing it can also be reduced, since it contains the necessary core verb and subject. The clause is reduced as an *adverbial modifier* that can also be reduced as complete. The final reduction will reassert the original *main clause* recognizer from which the others were generated, and the bindings created during the reductions will be left on its stack. One binding (from the core verb of the clause) will be left as a half-binding, anticipating its completion when the modified object (in this case, the core verb of the main clause, *know*) is parsed.

5.3. The Weight System

A syntactic recognizer carries a weight that shows its relative likelihood of being the correct parse of the input. When the entire input has been processed, the system selects the recognizer with the highest weight as the true parse. The weight is actually the sum of two separate weights, the first derived from the manipulations of the shift-reduce rules, the second from the activities of the semantic recognizers.

Experimentation has shown that there should be two separate weight values. Perhaps this is because the range of syntactic knowledge is relatively finite, whereas the range of semantic knowledge is undefined. When the syntactic function of a particular constituent is sought, one can usually be confident in forming an inclusive list of alternatives. This is not true of the semantic aspects of the constituent.

Syntactic weight values range from 0 to 100. The value represents the percentage of the time that, of the inclusive list of alternatives, a given choice actually occurs. A zero value would represent "never," and 100 "always." The initial *main clause* recognizer from which all

other syntactic recognizers are spawned is set to 100. The final values will tend to shrink from this high, although there will be both rises and falls during the course of parsing. Updating of accumulated syntactic weight values with a new weight value is performed by effecting the average of all previous weights with the new one.

Semantic weights range from approximately -100 to 100, although they may exceed this if necessary. Generally they are below 40. The initial *main clause* recognizer is assigned a semantic weight value of 0, and is expected to grow when selected syntactic patterns are reinforced by updating with semantic weights. The semantic weight represents the degree by which the accumulated weight value should be increased or decreased. A zero value here means no "no change."

The weights have been figured by hand, and adjusted as necessary. The initial values chosen for syntactic weights have required little refinement, but the semantic weights have needed more attention. When new semantic predicates are added to solve problems such as the proper attachment of trailing prepositional phrases, these must fit with other semantic predicates affecting the same elements. The evidence seems to show that the use of semantic weight values is a crucial part of producing the correct parse.

An important feature of the double weight scheme is that it potentially can have the power to handle special problems. When the input is grammatically incorrect and the parser may have a syntactic dead end, the semantic weights may still be able to select preferred alternatives. On the other hand, when nonsense elements occur, or unknown words, the syntactic weights alone may provide the answer.

The main question about the weight system is whether or not it will stabilize when sufficient knowledge has been implemented in the system. At this time the question cannot be answered, but there is no reason as yet to think that it will fail to stablize. Additions to the weight system have continued to be assimilated well.

5.4. Lexical Preprocessing

Preprocessing can provide several benefits. Most importantly, it can rewrite terms with an equivalent standardized term. The standardized term is maintained with complete semantic information in the lexicon. This saves space in the lexicon, and reduces the number of checks that must be performed during parsing. Secondly, preprocessing should respell contractions, abbreviations, certain symbols and perhaps digits. Also, it can provide information to direct or simplify parsing by checking for a small number of given phrases or strings.

In general, the philosophy has been adopted that preprocessing should be limited so that the parser itself can be tested more fully. This philosophy expects to trade the loss of time preprocessing could gain for greater generality in the parser itself.

At this time, the preprocessor contains about 200 respelling predicates. These include predicates that respell *line number* as *line_number* and *instead of* as *instead_of.* Only combinations that always have the same function are respelled. Common pairs, such as *right now* or *at a time*, are not since they may appear in a context where they function separately. For example, a sentence may begin "At a time when ..." Some respelling can be done that simply eliminates redundancies. UGURU's preprocessor respells *someone else* as *someone.*

The preprocessor also looks for key phrases that begin the kinds of questions that would be most commonly asked of UGURU. These are all "how to" questions, but may be phrased in a variety of ways:

How can I ....
How do I ....
How can you ...
How does one ...
What is the command to ...

All of these, and some others are replaced by the special token *howcani.* This greatly increases speed in parsing these questions. *I* and *you* lose their normal meaning in the context of user-software discourse. The simplification gained by equating them here, and in a couple of other contexts, has been advantageous.

5.5. Extent of Parallelism

One of the most obvious and crucial issues for the parser is whether the parallel pursuit of all parsing continuations will overwhelm the system, that is, whether the number of recognizers will become too great. As the parse continues only syntactic recognizers are ever eliminated, either through failure to match shift rules or deassertion by semantic recognizers. As the system accumulates knowledge through the installment of more shift-reduce rules and lexical entries that generate semantic recognizers during the parsing cycles, it also magnifies the number of possibilities. The question arises whether the system can tolerate the addition of new knowledge only up to a point, and past that point decisions become difficult because of the number of alternatives.

To date, the parallelism is manageable. It appears that the fact that new knowledge expands the search space is balanced by the constraints of the knowledge to some degree at least. Most general knowledge is already included in the system. New knowledge added now is more specific in nature, referring to specific words or bindings. Furthermore, most of the new knowledge is semantic in nature and provides information for making choices between alternatives using weights.

Inevitably, a great deal of tracing has been done to understand how the system is performing and to debug it. The traces show that the number of syntactic recognizers tends to rise and fall on most sentences in a naturally controlled manner due to the syntactic constraints. The high number of syntactic dead ends that occur at certain points in a sentence tend continually to reduce the accumulation of recognizers to a fairly small number, no matter how much larger the peaks are becoming with the addition of new knowledge. Also, increased accuracy with the system of weights would allow pruning to be done at every parsing cycle.

5.6. The Parser and Several Important Problem Areas of English

The following sections briefly describe the approach to certain problems of English taken in the parser design.

5.6.1. Conjunctions and Comparisons

One of the payoffs for the implementing the *object* level of constituents is the capability of handling conjunctions in a consistent manner. Design of this system has assumed that there is a principle of English that defines the use of conjunctions as the connection of two of the same type of *object*. Examples are

How can I compile my program quickly and without saving the warnings?

where two *adverbial objects* (*quickly*, *without saving the warnings*) are joined, and

I would like to combine two files and print them out.

where two *noun objects* (*to combine two files*, *[to] print them out*) are joined. Both of these examples present difficulties to a theory of conjunctions, unless the two objects can be considered as similar *objects*. In the second case, knowledge of the kind of *object* that ought to follow the conjunction can lead to help in solving the problem of the elliptical *to*.

To illustrate how managing conjunctions is implemented, consider a sentence with the phrase *hook, line and sinker*. Since every possible reduction must be made, *hook* is reduced as a *noun object* at the end of its parsing cycle. Eventually syntactic dead ends leading from the reduction will destroy it. *Hook* is also stored as a completed *noun phrase* within an incomplete *noun object*, which is indicated in the recognizer's state parameter as follows:

obj(noun)//0

The 0 shows that no punctuation has been found so far with the *noun object*, but this will be changed at the next parsing cycle when the comma is read. Subsequently, the *noun object* can only be reduced when a completed *noun phrase* is on top of the stack (the last item seen) and

an **and** or **or** has replaced any commas. This prevents the comma from being used as a conjunction all by itself.

Comparison operators can be handled in a manner similar to conjunctions, since they also require that corresponding *objects* be found. The case is more difficult since ellipsis is more frequent with comparisons. Comparison operators and conjunctions appear in the knowledge representation scheme as half-bindings. This has proved to be both a flexible and powerful way to incorporate them.

5.6.2. Ellipses and Indirect Reference

The system attempts to handle ellipses through knowledge of common cases formulated in the semantic recognizers. This is a severe limitation at this time, since it has been found very difficult to generalize how and where ellipses may occur. The problem of partial sentences occurring in sustained discourse has not been considered. The most important case that has been dealt with is relative clauses that begin elliptically. For example,

Where is the notice students are all talking about?

Unless the system knows that *that* can be added between *notice* and *students*, it will ultimately find a this sentence a syntactic dead end. Among its efforts, it will try to treat *notice students* as a compound noun, but this will not succeed. A semantic recognizer looking for adjoining nouns can supply the information and produce an appropriate new syntactic recognizer in which *students* is the first token in a *relative clause*.

Another situation treated in a similar manner is an input containing a sentence and a question. The system considers the sentence as information that is preconditional to the question, and so it supplies the "missing" *if* at the beginning of the sentence. The sentence becomes a subordinate clause in the question.

Indirect references do not present a particular syntactic problem, but the identity of the referent must be established so that all relationships in the statement can be clear.

Occasionally, proper attachment of prepositional phrases depends on the identity of an indirect reference contained in the phrase. As with ellipses, semantic recognizers should be used to resolve the reference, and at the point they occur, since the reference will almost certainly have been stated already. This is an area that has been only slightly implemented so far.

5.6.3. Compound Nouns

Compound nouns are handled both by shift-reduce actions and by semantic recognizers. Any two successive nouns can be absorbed as a potential compound noun by a shift rule, while reduction of the pair as a compound noun depends on a successful call by the reduce rule to a semantic predicate that checks the relationship of the two nouns. The main properties that are considered are **cmptof** (component of) and **instof** (instance of). Either of these relationships may exist in either order (see Section 4.3.1).

Cases of noun combination that cannot be generalized as well as **instof** or **cmptof** can be handled easily by semantic recognizers. Since semantic recognizers have the capability of thoroughly examining the targeted syntactic recognizer, they can also take into consideration the context of the potential compound noun. The capability to provide conditional acceptance makes the semantic recognizers powerful tools in this case, as in the situations described in the previous sections.

CHAPTER VI

THE RETRIEVER

Once the parser has interpreted the input as a set of bindings, it asserts them into the data base and marks them either as a question or a statement. A statement becomes part of the data base of Unix knowledge, and UGURU can then reinitialize itself to accept another user input.

A question causes the retriever to be invoked. The retriever's function is to find an answer to the question by searching the knowledge base for a matching set of bindings. The matched set also contains bindings that represent the information missing in the question. If the input question were

How can I see who's on the system right now?

the retriever would seek a representation of a statement equivalent to

You can see who's on the system right now by using the command who.

The meaning of *equivalence* for two sentences is the essential problem faced by the retriever. The manner in which the user phrases a question is outside the control of the program, and so the retriever must be able to map numerous rewordings of the same question to the single answer. For example, when the input question is

How can I find out who's working on the system?

the retriever should again locate the same fact above suggesting the use of the command who.

Actually the equivalence problem is *relativistic*, because, from the perspective of a given question, it is the fact that may appear to have been reworded. The alternative is to represent all pieces of knowledge in standardized forms through the use of a set of primitives that cover the domain of word meanings. In designing UGURU's knowledge representation this alternative was deliberately avoided. Most importantly, it does not appear to be psychologically realistic, which puts it in conflict with one of the UGURU's major design principles, modeling human language processing. Secondly, the representation schemes that employ this idea have so far yielded only crude results when applied to a large domain of knowledge. This issue will be considered at more length in Chapter VIII.

Since the only data structure the retriever sees is sets of bindings, matching inputs and facts is really the problem of the equivalence of the two sets of bindings. It accomplishes this by attempting to *transform* one set into the other set. The first example question and the fact above are directly equivalent in all bindings but one since their wording is identical. *I* in the question and *you* in the fact both obviously refer to the user. The retriever will finally also match *How* in the question to *by using the command who* in the fact. Both are *adverbial objects* that are bound to the verb *see* as its **means**. The retriever understands that *how* is a nonspecific place marker that matches anything whose binding relationship is equivalent. In general, any interrogative word can serve as a nonspecific place marker.

6.1. Knowledge Representation

Figure 6.1 contains the set of bindings generated by the parser for the input

How can I recover a file accidentally lost when the system crashed?

The list of bindings shown comprise the binding set produced by the parser from the question. Each binding is a triple consisting of two tokens from the input and the relationship between them. The binding also includes a *fact number* common to the binding set: an identical fact number unifies all the bindings that represent a single idea. The fact number for a question is the non-numeric value **q**, but all the facts - statements - that reside permanently in the knowledge base have integer fact numbers. Naturally this is convenient for bookkeeping purposes.

The relationships within a binding are indicated by the user-defined Prolog binary operators ##, >---, and --->. ## joins the fact number to the binding triple; it has the highest precedence of the three operators involved, and therefore will be seen by the system before it inspects the makeup of the binding, which is effectively parenthesized.[1] The two

---

```
q ## how:1//adv(intrg) >--- advmod(means^_) ---> recover:4//vb(cvb^base).
q ## user:3//noun(sg) >--- auth(n) ---> recover:4//vb(cvb^base).
q ## recover:4//vb(cvb^base) >--- cvb ---> 0.
q ## file:6//noun(sg) >--- obj(n) ---> recover:4//vb(cvb^base).
q ## file:6//noun(sg) >--- obj(n) ---> lose:8//vb(cvb^pastpcp).
q ## accidentally:7//adv(_) >--- advmod(s) ---> lose:8//vb(cvb^pastpcp).
q ## system:11//noun(sg) >--- auth(n) ---> crash:12//vb(cvb^pastpcp).
q ## crash:12//vb(cvb^pastpcp) >--- advmod(cond^when) ---> lose:8//vb(cvb^pastpcp).
```

Fig. 6.1 Binding set for *How can I recover a file accidentally lost when the system crashed?*

---

[1] --->has higher precedence than >---. This is not an important factor in the immediate discussion, but helps to make the building of the links by the parser more convenient. When an adjective is encountered in a noun phrase - for example, *fast*, the half-binding **fast>---adjmod(s)**is placed on the stack. When the noun is recognized, the binding can be completed. Subsequently, the noun will be more immediate to view than the adjective.

half-arrow operators are intended to help visualize the binding. The word on the far left is a dependent of the word on the far right; generally the first is a clear modifier of the second. The binding relationship appears in the midpoint of the arrow. Half-bindings that exist in the stacks of syntactic recognizers during the course of parsing are created using just the left half-arrow, >---. The left half-arrow is also used to create permanent half-bindings that represent logical relationships between *objects* in the input sentence. These relationships include the usual conjunctions and also comparisons.

6.1.1. Relationship of Binding Sets and Trees

If one considers the arrows as edges in a DAG, it is easily seen that the verbs are the focal points of the graph. In the case of Figure 6.1, every binding points to a verb. If the bindings are separated into three groups by the verbs, as in Figure 6.2a, three graphs emerge, each of which is a tree structure. Each of the three graphs represents a single clause, and the

---

how:1//adv(intrg) >--- advmod(means^_) ---> recover:4//vb(cvb^base).
user:3//noun(sg) >--- auth(n) ---> recover:4//vb(cvb^base).
file:6//noun(sg) >--- obj(n) ---> recover:4//vb(cvb^base).

How [can] user [= I] recover [a] file

file:6//noun(sg) >--- obj(n) ---> lose:8//vb(cvb^pastpcp).
accidentally:7//adv(_) >--- advmod(s) ---> lose:8//vb(cvb^pastpcp).

[a] file [was] accidentally lost

system:11//noun(sg) >--- auth(n) ---> crash:12//vb(cvb^pastpcp).

[the] system crashed

Fig. 6.2a Binding set for the three clauses in *How can I recover a file accidentally lost when the system crashed?*

---

root of each one is its core verb.[2] Figure 6.2a omits two bindings that show relationships between the clauses themselves. The binding to 0 is made only from the core verb of the main clause, indicating its primary position relative to any other clauses. The binding from *crash* to *lose* connects those two clauses and is marked to show the conditional relationship between them. A connection exists from the first subordinate clause, *[a] file [was] accidentally lost*, to the main clause, through their common node, *file*, and so no separate binding is needed. Though the binding set for the entire question does not form a tree structure, the fact that it can be broken into clausal tree structures exhibits the principle built into the parser that clauses are embedded sentences. Figure 6.2b shows a DAG representation of the binding set in Figure 6.1 in which the three trees representing the three clauses in Figure 6.2a may be seen, together with their interconnection. The roots of the three subtrees are the verbs *recover*, *lose*, and *crash*.

If the dependents on a core verb have dependents of their own, i.e., modifiers, the representation of a single clause still forms a tree structure. The secondary dependents lie in the second generation of the tree. Figure 6.3 shows the binding set for the main clause of the example in Figure 6.1 as if it were phrased *How can I recover an output file*. Figure 6.3 also shows a simplified tree representation of the four bindings, in which the lexical information have not been duplicated. It should be clear that the binding set notation and the graph are logically identical.

This representation scheme can be applied with consistency to the whole range of questions under study. It has simplicity and orthogonality. Its flexibility stems at least in part, however, from its willingness not to sharply define all the binding relationships. **manner** and **means** overlap in their denotations, as do **loc** and **dest**, or **loc** and **src**. The retriever must be able to match the non-specific place marker *how* not only with facts structured with **means** (or even **manner**) relationships, but with equivalent facts structured with a main clause and a

[2]That the binding sets are organized around the verbs is consistent with the approach of a number of semantic network theories beginning with Simmons *et al.* [Simm 68] and Rumelhart and Norman [Rume 73].

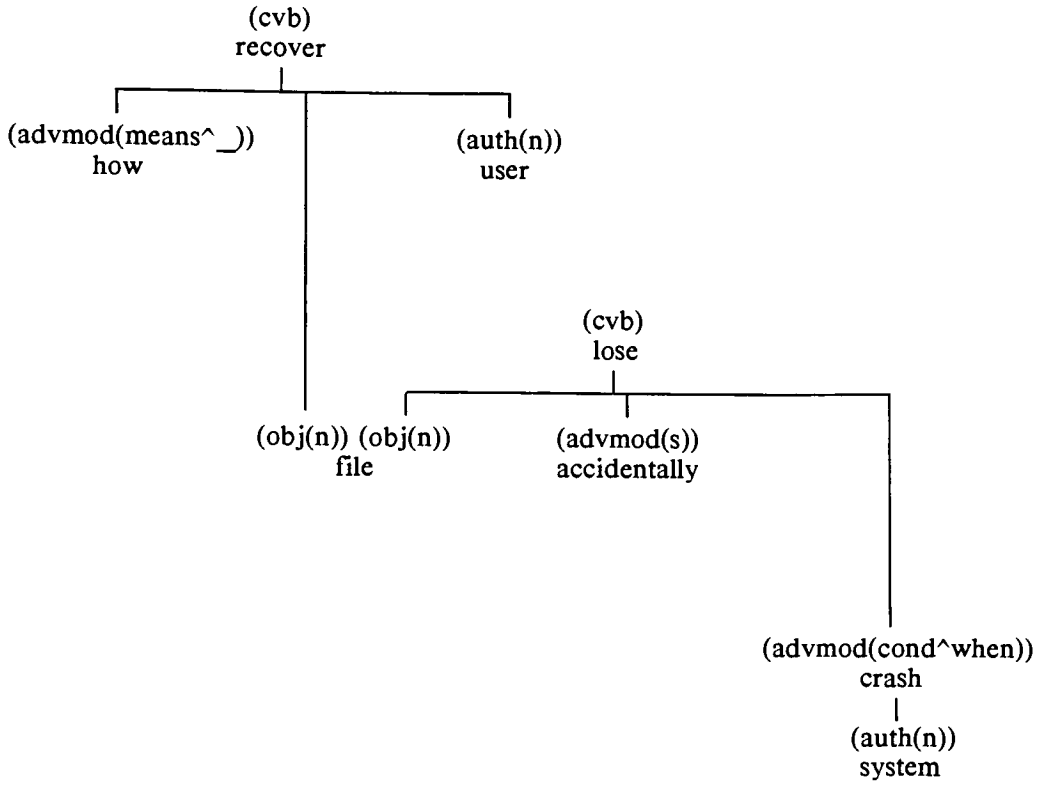

Fig. 6.2b DAG representation for *How can I recover a file accidentally lost when the system crashed?*

```
how:1//adv(intrg)  >---  advmod(means^_)  --->  recover:4//vb(cvb^base).
user:3//noun(sg)   >---  auth(n)          --->  recover:4//vb(cvb^base).
file:7//noun(sg)   >---  obj(n)           --->  recover:4//vb(cvb^base).
output:6//noun(sg) >---  instof(n)        --->  file:7//noun(sg).
```

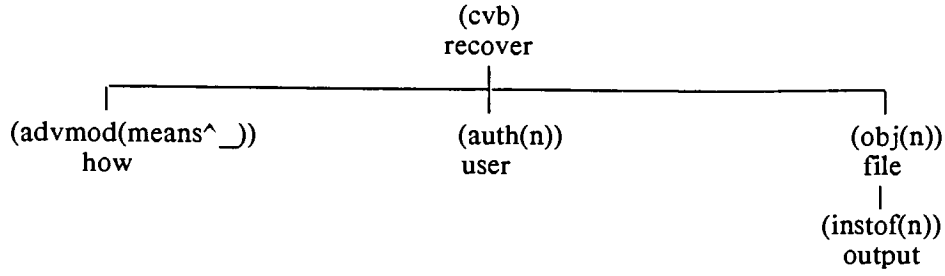

Fig. 6.3 Binding set and tree representation for
How can I recover an output file?

purp binding. Section 6.3 illustrates this point using the original example at the beginning of the chapter, *How can I see who's on the system right now?* Semantic knowledge regarding the bindings together with the words involved must sort out the cases where **means** and **manner** are equally applicable, and the cases where they are not.

The representation scheme also gains flexibility through the principle that syntactic constituents may substitute for constituents of another type. A simple example is the phrase *right now* that appears in the example question. The binding that joins *right* to *now* can only be allowed since the adjective can substitute as an adverb. The binding is in all other regards like a binding between two adverbs:

```
right:9//adj(_)  >---  advmod(s)  --->  now:10//adv(_).
```

A more substantial example is the substitution of a *clause* for a *noun object*. As the root of the subtree representing the *clause*, the clausal core verb is bound as a dependent of the core

verb of the main clause. The binding set for the statement

I would like to print out two files.

is listed in Figure 6.4. The example in Figure 6.4 demonstrates a couple of other points about the way in which the knowledge representation scheme works. In general, the parser does not save auxiliary verbs. In this case, however, *would* adds a degree of intentionality that shades the meaning of the statement. UGURU should treat this input as if it were a question, since the user really is asking for information. A fair number of examples of this type showed up in the surveys. The statement is clearly different from

I like printing out two files.

It is not a tenet of this system that two expressions of the same idea, differently phrased, must have the same representation. On the other hand, when the same stem words are used to express the idea, it seems natural that the representations also should be the same. In this matter, the system also performs well. The binding set for *I like printing out two files* is listed in Figure 6.5. It is identical to that in Figure 6.4 except that the binding of *would* is missing, and the form of *print* is recognized as **prespcp** instead of **base**. Both sentences contain a ver-

---

```
user:1//noun(sg)  >---  auth(n)  --->  like:3//vb(cvb^base).
would:2//vb(aux^base)  >---  aux  --->  like:3//vb(cvb^base).
like:3//vb(cvb^base)  >---  cvb  --->  0.
print:5//vb(cvb^base)  >---  obj(v)  --->  like:3//vb(cvb^base).
out:6//prep(partcl)  >---  advmod(dest^out)  --->  print:5//vb(cvb^base).
two:7//adj(num)  >---  adjmod(num)  --->  file:8//noun(pl).
file:8//noun(pl)  >---  obj(n)  --->  print:5//vb(cvb^base).
```

Fig. 6.4 Binding set for *I would like to print out two files.*

---

user:1//noun(sg) >--- auth(n) ---> like:3//vb(cvb^base).
like:3//vb(cvb^base) >--- cvb ---> 0.
print:5//vb(cvb^prespcp) >--- obj(v) ---> like:3//vb(cvb^base).
out:6//prep(partcl) >--- advmod(dest^out) ---> print:5//vb(cvb^prespcp).
two:7//adj(num) >--- adjmod(num) ---> file:8//noun(pl).
file:8//noun(pl) >--- obj(n) ---> print:5//vb(cvb^prespcp).

Fig. 6.5 Binding set for *I like printing out two files.*

bal clause used as a *noun object* in the same way. To further demonstrate the flexibility and consistency of the representation scheme, Figure 6.6 lists the binding set for

I would like the computer to print out two files.

The only difference between Figure 6.6 and Figure 6.4 is the addition of the binding between *computer* and *print*. In the statement *I would like to print out two files*, the parser does not explicitly create an **auth** binding for *print*, but a single semantic rule allows either the parser or the retriever to recognize that the **auth** of the core verb in the main clause can be carried

user:1//noun(sg) >--- auth(n) ---> like:3//vb(cvb^base).
would:2//vb(aux^base) >--- aux ---> like:3//vb(cvb^base).
like:3//vb(cvb^base) >--- cvb ---> 0.
computer:5//noun(sg) >--- auth(n) ---> print:7//vb(cvb^base).
print:7//vb(cvb^base) >--- obj(v) ---> like:3//vb(cvb^base).
out:8//prep(partcl) >--- advmod(dest^out) ---> print:7//vb(cvb^base).
two:9//adj(num) >--- adjmod(num) ---> file:10//noun(pl).
file:10//noun(pl) >--- obj(n) ---> print:7//vb(cvb^base).

Fig. 6.6 Binding set for *I would like the computer to print out two files.*

over as **auth** of the object verb, if it is unspecified. In the sentence *I would like the computer to print out two files*, the explicit binding of *computer* simply replaces the implicit binding of *I* (user). Otherwise, all the interclausal relationships remain the same.

The system also automatically produces the same binding set in most cases for sentences that are elliptical and for their more complete counterparts. For example, the binding set for

How can I recover a file that was accidentally lost when the system crashed?

is the same as its elliptical version in Figure 6.1 in which the words *that was* are omitted. During the course of parsing, the original **obj** of *lose* in this version is the pronoun *that*. The binding

file:5//noun(sg) >--- obj(n) ---> lose:7//vb(cvb^pastpcp).

could be generated directly from the elliptical version; in this case it will be produced when the parser attempts to find a link between the subordinate clause and the main clause, and substitutes *file* for *that*. The retriever will see identical binding sets.

6.1.2. Representation of Conjunctions and Comparisons

Completed binding sets at times include half-bindings. The half-binding relationships represent situations in which the language of the input uses the fundamental logical relationships to organize syntax. These are **and**, **or**, and **neg** (negation); also included in this group are **than** (comparisons). The treatment of the half-bindings is consistent with the other principles of binding set representations, and it maintains the tree structure of single clauses though horizontal connections are implied to exist between siblings. Except for **neg**, which is variously incorporated as a half-binding or a full binding, the logical relationships are binary. Since the parser implements the principle that there may be only a single instance of any binding relationship among the several between each of a set of siblings and their parent, these half-bindings allow more than one token to be mapped as a single instance of the particular binding relationship involved.

The half-bindings also observe the important principle that only objects of the same type can be connected. An *object* may comprise many words of an input statement, e.g., a conditional clause that functions as an *adverbial object.* Its representation, excluding its own subordinate clauses that are modifiers, must form a tree structure. The logical half-bindings connect two *objects* by connecting the roots of their respective subtrees.

As examples, first consider the question

What is the difference between Mail and mail?

Figure 6.7 lists the corresponding binding set and a tree representation. In the binding set,

---

```
what:1//pronoun(intrg)  >---  auth(n)  --->  be:2//vb(cvb^base).
be:2//vb(cvb^base)  >---  cvb  --->  0.
difference:4//noun(sg)  >---  trnsfl(n)  --->  be:2//vb(cvb^base).
Mail:6//noun(sg)  >---  adjmod(loc^between)  --->  difference:4//noun(sg).
Mail:6//noun(sg)  >---  and(1).
mail:8//noun(sg)  >---  adjmod(loc^between)  --->  difference:4//noun(sg).
mail:6//noun(sg)  >---  and(1).
```

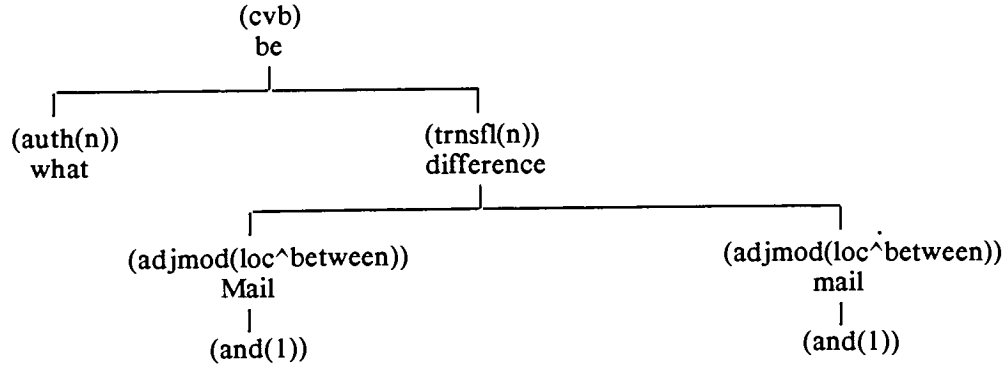

Fig. 6.7 Binding set and tree representation for
What is the difference between Mail and mail?

---

the half-binding **and** links *Mail* and *mail.* In the example, *difference* may have two dependents with identical binding relationships since the conjunction combines the pair as a single instance of the relationship. Figure 6.7 also shows that the relationship **and** is numbered; this is necessary to prevent confusion when more than one logical binding is present.

Note that the theme of comparison in this example remains in a deep level of sentence organization, a meaning level, and does not shape surface structures with the use of comparative adverbs or adjectives, or phrases such as *more than, less than, rather than,* or *instead of.* In this respect, the example demonstrates by default the syntactic nature of the logical half-bindings, because understanding that the question asks for a comparison depends entirely on knowing the meaning of the word *difference.* It is easy to produce questions that are syntactically identical to the example question in Figure 6.7, using the same verb and preposition, but which do not convey the theme of comparison. One such question is *Which is the road between Salem and Boston?* Some examples that use the **than** half-binding follow later in this section.

To illustrate the fact that the logical half-bindings connect subtrees and not just single tokens, Figure 6.8 lists the binding set and a tree representation for the question

Can I talk to someone with either the "talk" command or the "write" command?

The subtrees referred to are the representations of *the "talk" command* and *the "write" command.* These are minimal examples, but it should be clear that the subtree could be indefinitely large. This example also shows the reasons why tokens are counted as they appear in the input, and tagged with the *token:number* syntax: the numbers distinguish the two instances of the word *command* and the two instances of the verb *talk.*

Logical half-bindings may join more than two *objects* simply by adding the extra statements necessary. The binding set for the statement

The names of the available debugging commands are **adb, dbx** and **pdx.**

```
user:2//noun(sg)  >---  auth(n)  --->  talk:3//vb(cvb^base).
talk:3//vb(cvb^base)  >---  cvb  --->  0.
someone:5//pronoun(_)  >---  advmod(indobj^to)  --->  talk:3//vb(cvb^base).
talk:9//vb(cvb^base)  >---  instof(v)  --->  command:10//noun(sg).
command:10//noun(sg)  >---  advmod(means^with)  --->  talk:3//vb(cvb^base).
command:10//noun(sg)  >---  or(1).
write:13//vb(cvb^base)  >---  instof(v)  --->  command:14//noun(sg).
command:14//noun(sg)  >---  advmod(means^with)  --->  talk:3//vb(cvb^base).
command:14//noun(sg)  >---  or(1).
```

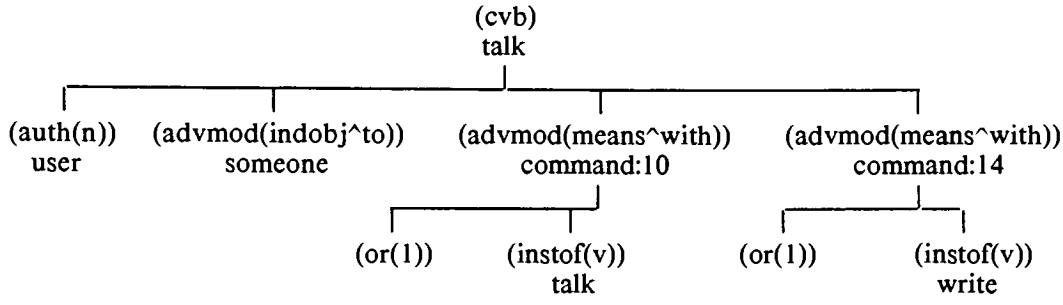

Figure 6.8 Binding set and tree representation for
Can I talk to someone with either the "talk" command or the "write" command?

contains links to **and(1)** from each of the three names, *adb*, *dbx*, and *pdx*. The parser binds each command name as a **trnsfl** of the core verb *is*, and will recognize that the comma in this case is a substitute for *and*.

This representation of conjunctions as half-bindings successfully covers a vast majority of the cases that have been considered for this project. There are two important limitations that require an extension of the simple means described. The first problem is the pairing of *objects* that have shared dependents. The most frequent occasion of this problem by far is the pairing of core verbs by conjunctions. In the question

the *objects* bound to *compile* must also be bound to *link*: *I* is the **auth** of both verbs, *program* the **obj**, and *how* the **means**. The parser is designed to duplicate all bindings in these cases. Extraction of the values from the knowledge base may only require one or the other of the conjoined *objects*, and therefore all dependent information must be present for each *object*. The binding set for the last example is shown in Figure 6.9. The representation pays a price in the cost of duplication of mutual bindings. However the cost is not as high as it might initially appear, since only the first generation bindings must be duplicated. If all information were to be extracted about the verb *compile*, the link that binds *C* to *program* would be included, since all information about the dependents of *compile* is required as well in order to complete the idea. The situation is symmetrical if the information about *link* is desired instead.

By duplicating mutual dependents, no clarity is lost in the representation of paired *objects* that have only some or perhaps none of their dependents in common. In the following example,

---

```
how:1//adv(intrg)  >---  advmod(means^_)  --->  compile:4//vb(cvb^base).
how:1//adv(intrg)  >---  advmod(means^_)  --->  link:6//vb(cvb^base).
user:3//noun(sg)  >---  auth(n)  --->  compile:4//vb(cvb^base).
user:3//noun(sg)  >---  auth(n)  --->  link:6//vb(cvb^base).
compile:4//vb(cvb^base)  >---  cvb  --->  0.
compile:4//vb(cvb^base)  >---  and(1).
link:6//vb(cvb^base)  >---  cvb  --->  0.
link:6//vb(cvb^base)  >---  and(1).
C:8//noun(sg)  >---  instof(n)  --->  program:9//noun(sg).
program:9//noun(sg)  >---  obj(n)  --->  compile:4//vb(cvb^base).
program:9//noun(sg)  >---  obj(n)  --->  link:6//vb(cvb^base).
```

Fig. 6.9 Binding set for *How can I compile and link a "C" program?*

How can I see just part of a file or send it to the printer?

see and *send* share the **auth** *I*, the **obj** *part* and the **means** *how*; *to the printer* must be bound only to *send*.

A somewhat less crucial problem in the half-binding scheme for representing conjunctions is its limited ability to show the nesting of logical relations. The problem is not crucial because it surfaces so rarely. A sentence like *Send the message to Jim and either Sally or Ted* suggests the possibility of the following bindings

Jim >--- and(n_1).
or(n_2) >--- and(n_1).
Sally >--- or(n_2).
Ted >--- or(n_2).

This example is included here to show that the half-binding scheme is capable of extension. However, such an extension is not warranted at present, and the system does not include the facilities for nesting logical relationships.

Comparisons manifested at the surface level by comparative suffixes or phrases such as *more than* are represented in a manner similar to the logical relationships. As pointed out in Section 5.6.1 a comparison relates two alternative *objects* competing for the same binding within a clause or phrase. The *objects* are the same constituent type; they differ with regard to some aspect of their meaning. Since people do not duplicate the shared constituents of the parent clause when they use language, this normally results in a fair amount of ellipsis. The situation regarding shared constituents is different with comparisons than with logical relations, since the shared *objects* here are siblings of the compared *objects*, rather than their dependents.

Figure 6.10 lists the binding sets for two of the examples of comparisons that were discussed in Section 5.6.1. The half-binding notation **than** is applied to compared *objects* as and is to conjoined *objects*. Each of the *objects* has a half-binding to the predicate **than**. An occurrence number is assigned as an argument to **than** like it is for the other logical relationships, but an additional argument is also required to show the direction of the comparison.

```
use:2//vb(cvb^prespcp)  >---  auth(v)  --->  run:7//vb(cvb^base).
use:2//vb(cvb^prespcp)  >---  than(1,more).
pc:3//noun(sg)  >---  instof(n)  --->  command:4//noun(sg).
command:4//noun(sg)  >---  obj(n)  --->  use:2//vb(cvb^prespcp).
command:4//noun(sg)  >---  and(1).
a.out:6//noun(sg)  >---  obj(n)  --->  use:2//vb(cvb^prespcp).
a.out:6//noun(sg)  >---  and(1).
run:7//vb(cvb^base)  >---  cvb  --->  0.
quickly:9//adv(comprv)  >---  advmod(s)  --->  run:7//vb(cvb^base).
pix:11//noun(sg)  >---  auth(n)  --->  run:7//vb(cvb^base).
pix:11//noun(sg)  >---  than(1,more).
```

Fig. 6.10a Binding set for *Does using the pc command and a.out run more quickly than pix?*

```
how:1//adv(intrg)  >---  advmod(means^_)  --->  print:4//vb(cvb^base).
user:3//noun(sg)  >---  auth(n)  --->  print:4//vb(cvb^base).
print:4//vb(cvb^base)  >---  cvb  --->  0.
listing:6//noun(sg)  >---  instof(n)  --->  file:7//noun(sg).
file:7//noun(sg)  >---  obj(n)  --->  print:4//vb(cvb^base).
file:7//noun(sg)  >---  than(1,more).
source:11//noun(sg)  >---  instof(n)  --->  file:12//noun(sg).
file:12//noun(sg)  >---  obj(n)  --->  print:4//vb(cvb^base).
file:12//noun(sg)  >---  than(1,less).
```

Fig. 6.10b Binding set for *How can I print the listing file rather than the source file?*

Of the two *objects* related by **than**, one must always be **more** and one **less**. Normally, the first stated is **more**, the second **less**. In Figure 6.10a, *using the pc command and a.out* is stated before the key word *than*, and *pix* follows. The half-binding of *use* (the root of the subtree representing the first *object*) to **than** contains the value **more**, which encodes the idea *runs more quickly*, while *pix* is bound to **than** with value **less**, standing for *runs less quickly*. In Figure 6.10b, the **more** and **less** values can be interpreted that *listing file* is preferred to *source file*. The situation is reversed if the comparison is formed with the phrase *less than*. The binding set for *Does pix run less quickly than using the pc command and a.out?* is identical to the binding set in Figure 6.10a. The fact that the actual bindings and half-bindings are

produced in a different order for the two versions of the question does not affect the sense or the use of the binding set as a whole.

The shape of the trees that correspond to binding sets containing **than** links have the same properties of those with logical relationships. In some more complicated comparisons, those with a fair amount of ellipsis, for example, the duplication of edges does become necessary for clarity, as with the cases of two core verbs joined by a conjunction.

Since the **neg** half-binding is unary, it is the simplest of the logical relationships to understand in the context of binding sets and the tree structures inherent in them. If the input sentence contains a negative word, such as *not*, *never*, or *hardly*, **neg** will bind the actual negative word to the word it modifies in a full binding. The **neg** half-binding results from implied negation. A primary example of implied negation is the use of the preposition *without*, as in the following question:

How can I repeat a command without retyping the whole thing?

The binding set for this questions is shown in Figure 6.11. The parser interprets this question as a main clause, *How can I repeat a command*, and a negative conditional clause

---

```
how:1//adv(intrg)  >---  advmod(means^_)  --->  repeat:4//vb(cvb^base).
user:3//noun(sg)  >---  auth(n)  --->  repeat:4//vb(cvb^base).
repeat:4//vb(cvb^base)  >---  cvb  --->  0.
command:6//noun(sg)  >---  obj(n)  --->  repeat:4//vb(cvb^base).
retype:8//vb(cvb^prespcp)  >---  advmod(cond^without)  --->  repeat:4//vb(cvb^base).
retype:8//vb(cvb^prespcp)  >---  neg.
whole:10//adj(_)  >---  adjmod(s)  --->  thing:11//noun(sg).
thing:11//noun(sg)  >---  obj(n)  --->  retype:8//vb(cvb^prespcp).
thing:11//noun(sg)  >---  ref  --->  command:6//noun(sg).
```

Fig. 6.11 Binding set for *How can I repeat a command without retyping the whole thing?*

---

reworded as *if I don't retype the whole thing*. The negation of *retype* is implied by the word *without*, and so the binding set will include a negative half-binding from *retype*.

6.2. The Knowledge Base

The knowledge base is the accumulation of binding sets that represent facts about the Unix environment. Each binding set has a fact number that identifies each of its member bindings, and keeps the distinction between one binding set and another.

UGURU's knowledge includes, of course, the knowledge needed by the parser. In fact, the knowledge required by the retriever overlaps with that of the parser. Arguably, all knowledge should eventually be available to both functions in the best of all worlds. At this stage of development, categorizing words into their classes and components is common to both. For example, the parser currently uses the format

instof(pascal, language).

to show that *pascal* is an instance of a *language*. The knowledge base available to the retriever can contain binding sets that include the member binding

pascal:*n*//noun(sg) >--- instof(n) ---> language:*m*//noun(sg).

meaning exactly the same thing. The parser's notation could be rewritten throughout the system to coincide with that of the retriever without changing its other predicates or procedures. Since the first notation is simpler and requires less overhead, the change should be made when the advantage outweighs the cost. This point would occur when the parser is expanded to include semantic recognizers that inspect the stored binding sets, that is, the entire knowledge base, looking for contextual knowledge that further clarifies the problems of parsing. This is possibly only one step away from the current state of the parser, since the semantic recognizers already represent the objects of their knowledge in terms of bindings and the arrow notation.

One of the important features of the representation of knowledge through binding sets is the independence of individual bindings. Although currently each binding set is asserted as

physically separate entity distinct from all other binding sets, this need not be so. The order in which bindings are asserted is, of course, unessential to the accuracy of the representation. This is true whether just the bindings within a single set were rearranged or the knowledge base as a whole were reordered. When all bindings identified by a single fact number are collected, the interrelationships of all parts is unambiguous: there is only one corresponding DAG.

There are two important advantages the system gains from an order-independent representation. First, and most important, it allows the comparison of two binding sets through the simple retrieval of the appropriate parts. Prolog is admirably suited to make this feature all the more efficient since it stores each binding through hashing. Procedurally, the retriever continually needs to find subgraphs in the binding sets it is comparing. It begins with a single token or binding, and locates the remaining bindings for a DAG of which the original provides the root. Each binding can be located directly through Prolog's hashing functions.

In a sense, retrieving a subgraph corresponds to retrieving a part of the idea represented by the whole binding set. On the other hand, every individual binding represents an idea. Defining *idea* is a metaphysical pursuit that is not an objective in this project, but the fact that it is so difficult argues the fact that it is possibly wrong to try to specify exact boundaries. The second advantage in the order-independent representation scheme used in UGURU is that, while each binding set is now a physically separate set of assertions, it is possible to rewrite the knowledge base in a way to save physical space without changing the logical organization. A single binding may appear as a member in numerous binding sets; this may reasonably portray the fluid aspect of the boundaries between ideas. An alternative way of declaring the data base is to declare each binding once, and assign it not a single fact number, but a list of fact numbers indicating the binding sets of which it is part. The question is whether a savings in space outweighs the extra processing time involved with the fact number list. This question can only be answered when the system is fully implemented. However, the new physical arrangement more clearly points up the

potential of the system to treat ideas as in an associative memory model. Since this is a function of the logical organization of the knowledge base, the current version has the same capabilities; it is possibly less obvious. Classes of words and ideas can be generated in a natural way either by spinning off from a token or set of tokens, or a fact number or set of fact numbers.

An important characteristic of the knowledge base is that the wording of the original statement expressing a piece of knowledge determines the binding set produced for it. This puts the responsibility on the retriever to be able to match statements or questions that rephrase the original. The pros and cons of this position will be put off until Chapter VIII. It has already been asserted that the forced reduction of the knowledge base to a set of primitive values was not considered appropriate for an NLP system. A second reason for approaching the system design in this way was to allow easy addition of knowledge by the user himself. It would naturally be difficult to control the way facts must be expressed if there is general access to enlarging the data base. The system could not permit general access to the permanent data base residing with the system code. However, by creating local files within the individual user's account or setting a pathway to special group-accessible files, the user would be able to store conveniently pieces of knowledge for his own recall later or for sharing with others. The surveys of questions led directly to the goal of a user-accessible knowledge base, since many of the questions were concerned with finding time-saving tricks that would be used fairly infrequently. Once a user discovers one such convenience, it would be valuable for him to be able to store it in a way that the information could be retrieved by himself or others with a natural language query.

6.3. Transforming Sets of Bindings

The retriever's task is to take a binding set with fact number q, representing a question processed by the parser, and find an *equivalent* binding set among the facts in the knowledge base, all of which are integer numbered. Equivalence implies that the two binding sets have the same *meaning*, but represent distinct surface level transformations. In most cases, the q-

numbered binding set, or **q**-set, will contain a binding from its main clause core verb to a nonspecific interrogative place marker, such as *how*, *what*, *when*, and so forth, that represents the goal of the question. Sometimes an interrogative is not present, and the system assumes that the question is asking for a verification of some fact, e.g, *Is DIR a Unix command?*

The retriever's first step is to locate the binding in the **q**-set with the interrogative place marker. It reserves this binding, and begins a series of actions aimed at matching the remainder of the **q**-set, to a subset of an integer-numbered target binding set. If such a match is found, the unmatched remainder of the target set should consist of a subtree-like structure of bindings whose root has the same binding relationship to the core verb in the target as the place marker has in the **q**-set. At the beginning of this chapter, the question

How can I see who's on the system right now?

was presented as an example, together with the sample answer

You can see who's on the system right now by using the command **who**.

Figure 6.12 shows the binding sets for the question and for the answer, or fact. The **q**-set generated from the question contains a binding with the nonspecific place marker *how*, which is starred to represent its being reserved during the course of the matching sequence performed by the retriever. The target set produced from the fact contains a subtree representing the clause *by using the command who*, and the bindings forming the subtree are also starred. The non-starred bindings in the **q**-set and the target otherwise match precisely, with the exception of the occurrence numbers. It is the non-starred bindings in the **q**-set for which the retriever first locates a match. When this particular target has been found, the retriever recognizes that *how* in the **q**-set and *use* in the target are bound by identical relationships to the main clause core verb. This satisfies the retriever's conditions for equivalence, and it will output the starred target subtree as the answer to the question.

* how:1//adv(intrg) >--- advmod(means^_) ---> see:4//vb(cvb^base).

user:3//noun(sg) >--- auth(n) ---> see:4//vb(cvb^base).
see:4//vb(cvb^base) >--- cvb ---> 0.
who:5//pronoun(intrg) >--- auth(n) ---> be:5//vb(cvb^base).
be:5//vb(cvb^base) >--- obj(v) ---> see:4//vb(cvb^base).
system:8//noun(sg) >--- advmod(loc^on) ---> be:5//vb(cvb^base).
right:9//adj(_) >--- advmod(s) ---> now:10//adv(_).
now:10//adv(_) >--- advmod(s) ---> be:5//vb(cvb^base).

q-set for *How can I see who's on the system right now?*

user:1//noun(sg) >--- auth(n) ---> see:3//vb(cvb^base).
see:3//vb(cvb^base) >--- cvb ---> 0.
who:4//pronoun(intrg) >--- auth(n) ---> be:4//vb(cvb^base).
be:4//vb(cvb^base) >--- obj(v) ---> see:3//vb(cvb^base).
system:7//noun(sg) >--- advmod(loc^on) ---> be:4//vb(cvb^base).
right:8//adj(_) >--- advmod(s) ---> now:9//adv(_).
now:9//adv(_) >--- advmod(s) ---> be:4//vb(cvb^base).

* use:11//vb(cvb^prespcp) >--- advmod(means^vb) ---> see:3//vb(cvb^base).
* command:13//noun(sg) >--- obj(n) ---> use:11//vb(cvb^prespcp).
* who:14//pronoun(intrg) >--- instof(n) ---> command:13//noun(sg).

Target set for *You can see who's on the system right now by using the command who.*

Fig. 6.12 Correlation of q-sets and target sets

To answer the fundamental question, are two binding sets equivalent, the retriever uses semantic predicates to transform one set into the other. To do so, it breaks the matching problem up into parts. It first seeks equivalent core verbs. Then it tries to match each subtree dependent on one core verb with a dependent subtree on the other core verb. Matching each subtree proceeds in the same manner, first matching its root, then matching its subtrees. In the simplest cases, demonstrating equivalence is a recursive procedure, similar to an infix traversal. The retriever solves the equivalence of the whole by showing the equivalence of

corresponding nodes and bindings. The triple contained in a single binding is the smallest unit to which transformations are applied. Bindings are equivalent if their components are the same or synonyms. The simplest binding set transformations are those where a one-to-one correspondence exists between the bindings of the **q**-set and the target. Generally, this is not the case, and the retriever may use the semantic predicates with either of two strategies: it may show that one or more bindings are unessential and disregard them, or it may match an odd number of bindings in one set with a different number in the other set.

In processing the example in Figure 6.12, the retriever works from the **q**-set. It reserves the binding from *how* to *see*. Then, referencing the knowledge base of facts, it locates a core verb binding to match the one in the **q**-set. Of the dependent bindings to *see* in the **q**-set, it can easily match its **auth** *user* to the **auth** *user* in the target. Because the order of bindings is unessential to the representation scheme, all that is required is that each dependent must be matched in some order. The retriever generally follows the order in the **q**-set. The only other child of *see* in the example **q**-set is the subtree representing the clause *who's on the system right now*. In matching an equivalent clause in the target, the retriever uses the same strategy of infix traversal. First to be matched in this case, is the core verb of the clause, *be*, since, as **obj**, it is bound as a direct dependent of *see*. Successively, matches will be found in the target for *who* and *system*, which do not have dependents of their own. Matching *now* requires matching its child *right* also.

When the equivalence of the **q**-set minus the reserved binding to bindings in the target is established, the retriever checks that the remaining bindings in the target form a single subtree that is bound to the core verb just as *how* is in the **q**-set. It concludes by outputting *Use the command who*, the information sought by the input.

The foregoing example is exceptional, of course, in its simplicity and the identical phrasing of the question and the fact. There are numerous complications that arise from the fact that the question and a corresponding fact may be reworded relative to one another. If the rewording only reorders the same words, the processing as described above would be

undisturbed. *By using the command who, you can see who's on the system right now* generates exactly the same binding set as *You can see who's on the system right now by using the command who.*

When the rephrasing employs different words, the retriever calls upon a base of semantic information to resolve the differences. In the program this constitutes a large collection of predicates simply called **wordsame** and **edgesame**. Like the semantic recognizers in the parser, these predicates can provide a range of levels of information. In both cases, a major goal of the system's design was to create a single mechanism with the extensibility to capture as wide a variety of analysis as possible. The retriever's semantic predicates include knowledge of a simple semantic nature, such as synonyms or classes, though knowledge that can be characterized as word constraints does not appear, since the parser's use of word constraints should already have eliminated unacceptable associations of words. The **edgesame** predicates also must supply information of an inferential or pragmatic nature, as the following examples will show.

The easiest of the equivalence problems to solve is synonym substitution. If the question in example 6.12 were rephrased as

How can I find out who's on the system right now?

equivalence can easily be established once *find out* is recognized as a synonym of *see*, still the core verb in the corresponding fact. One of the first objections that may be given to the generality of this method of synonym substitution is that *see* has meanings that do not correspond to *find out*, and vice versa. The system design assumes that, if the equivalence of the remaining bindings in both the **q**-set and the target is established, this adequately guarantees that the synonym relationship has been applied properly. This is true even when there are several synonym mappings in parallel between the **q**-set and the target. It is possible to generate large numbers of nonsense sentences through the misuse of synonym substitution, but each binding set processed by the retriever can already be assumed to have a coherent set of rela-

tionships holding it together, either because it is the output of the parser, or it was coded directly into the knowledge base of facts.

Mapping *find out* to *see* illustrates one of the logistical problems involved in the execution of synonym transformations. In this case, the retriever cannot transform the two only by looking at the core verbs *find* and *see*. The semantic predicates check to see if *find* has a childless dependent *out*. The subtree *find* plus *out* is equated with the single node *see*. The retriever frequently performs transformations of this sort, where a number of bindings in one binding set are equated with an unequal number of bindings in the other set. An alternative solution for this particular case derives from preprocessing performed on the input. The preprocessor simplifies the input by creating a single token, *find_out*, for the two words *find* and *out*. A synonym entry for *see* and *find_out* allows the retriever to transform the core verbs in a single step. Except for the limited number of cases where the preprocessor replaces idioms of several words with a single token (see Section 5.4), their use normally causes the retriever to match several bindings in one binding set with an unrelated number of bindings, frequently one, in the other set.

Disregarding bindings that are not essential to the idea associated with a binding set is another important ability the retriever gains from the semantic predicates. *How can I see who's on the system right now?* might be rephrased as

How can I see who's presently on the system?

Figure 6.13 lists the adjusted **q**-set. The new wording requires a transformation of the binding containing *presently* in the **q**-set generated for the quote above to the two bindings involving *right* and *now* in the target binding in Figure 6.12. Again, there are two ways this can be handled. Preprocessing could create a single *right-now* token; in fact, the system does not do this (see Section 5.4). The alternative is to provide a semantic predicate that recognizes that *right* is redundant in this situation. Together with **wordsame** and **edgesame**, the retriever uses another group of **insignif** predicates to identify cases similar to the example here.

how:1//adv(intrg) >--- advmod(means^_) ---> see:4//vb(cvb^base).
user:3//noun(sg) >--- auth(n) ---> see:4//vb(cvb^base).
see:4//vb(cvb^base) >--- cvb ---> 0.
who:5//pronoun(intrg) >--- auth(n) ---> be:5//vb(cvb^base).
be:5//vb(cvb^base) >--- obj(v) ---> see:4//vb(cvb^base).
presently:6://adv(_) >--- advmod(s) ---> be:5//vb(cvb^base).
system:9//noun(sg) >--- advmod(loc^on) ---> be:5//vb(cvb^base).

Fig. 6.13 q-set for *How can I see who's presently on the system?*

The ways in which the **insignif** predicates are used to enable transformations are actually quite numerous, and more complex than the example above. Another example can serve to illustrate this further. The system contains a binding set for the fact

> The user can send mail to someone on the system by using the **mail** command with the person's login name.

The retriever must match a **q**-set generated from the input question

> How can I mail a message?

The binding sets are shown in Figure 6.14. The core verb of the **q**-set, *mail*, has three dependents. Its **auth** is *user (I)*; its **obj** is *message*; its **means** is the nonspecific place marker *how*. The core verb of the target is *send*, which has four dependents. Its **auth** is also *user*, the same as the **q**-set. Its **obj**, however, is *mail*; its **indobj** is *[to] someone [on the system]*; its **means** is *[by] using [the mail command with the person's login name]*. The retriever must first match core verbs. Semantic predicates establish that *mail* in the **q**-set is equivalent to *send mail* in the target. The **means** in the **q**-set is temporarily reserved, but will eventually match the **means** clause in the target. This leaves an unmatched **obj** binding in the **q**-set, from *message* to *mail*, and an unmatched *indobj* binding in the target, from *to someone on the system* to *send*.

```
how:1//adv(intrg) >--- advmod(means^_) ---> mail:4//vb(cvb^base).
user:3//noun(sg) >--- auth(n) ---> mail:4//vb(cvb^base).
mail:4//vb(cvb^base) >--- cvb ---> 0.
message:6//noun(sg) >--- obj(n) ---> mail:4//vb(cvb^base).
```

Fig. 6.14a q-set for *How can I mail a message?*

```
user:2//noun(sg) >--- auth(n) ---> send:4//vb(cvb^base).
send:4//vb(cvb^base) >--- cvb ---> 0.
mail:5//noun(sg) >--- obj(n) ---> send:4//vb(cvb^base).
someone:7//pronoun(indef) >--- indobj(n) ---> send:4//vb(cvb^base).
system:10//noun(sg) >--- adjmod(loc^on) ---> someone:7//pronoun(indef).
use:12//vb(cvb^prespcp) >--- advmod(means^vb) ---> send:4//vb(cvb^base).
mail:14//noun(sg) >--- instof(n) ---> command:15//noun(sg).
person:18//noun(sg) >--- adjmod(possv) ---> name:20//noun(sg).
login:19//noun(sg) >--- instof(n) ---> name:20//noun(sg).
name:20//noun(sg) >--- advmod(near^with) ---> name:20//noun(sg).
```

Fig. 6.14b Target set for *The user can send mail to someone on the system by using the mail command with the person's login name*

insignif predicates determine that *message*, if unmodified, is redundant, and, therefore, since it is unessential, a match is not required in the target. Likewise, *to someone on the system* can also be disregarded since it also does not provide any essential information beyond the other bindings in the target. The retriever considers the **q**-set and the target equivalent, because all essential bindings have transformations from one set to the other.

The situation is different for the example in Figure 6.14 if either the question or fact, or both, are reworded to include a reference to *another system.* The correct answer for this question will specify the *mail* command and also how to write the complete path for addressing persons on other systems, whereas the original question, *How can I mail a message*,

should retrieve the more general answer.[3] The addition of *another* as a modifier for *system* makes the *indobj* binding in the target an essential piece of information, and the q-set as given is no longer a transformation of the target since there is not a matching binding conveying the idea of *another system.*

Returning to the example in Figure 6.12, another type of transformational problem is presented by a rephrasing of the input question as

How can I see if John is on the system right now?

John is a specific example of *who* might be on the system. The retriever considers *John* and *who* equivalent through the same sort of syllogistic reasoning based on **instof** (instance of) class predicates used in the semantic recognizers. In this case, a *person* is a legitimate instance of *who*, a *person_name* is an instance of a *person*, and *John* is an instance of a *person_name.* There is one other change in the q-set for this version of the example question compared to the q-set in Figure 6.12. This change reflects the use of *if* to introduce the subordinate clause. *be* is now bound as the **cond** of the core verb *see* rather than as its **obj.** The transformation of one subordinate clause to the other is achieved by a semantic predicate that equates a **cond** clause and an **obj** clause containing an interrogative word when they are dependents of the class of verbs that includes *find_out* and *see.*

The examples so far have demonstrated transformations that are based on the abilities of the semantic predicates to find synonyms or to disregard unessential bindings. The semantic predicates also need sufficient power to handle transformations that depend on inferences about the meanings of the binding sets. An instance when this is needed is when a fact with a **means** binding is restated with a **purp** binding instead. The fact in Figure 6.12 would appear as follows:

---

[3]The objective in UGURU's design is to return as specific and narrow an answer as possible for each question. This minimalist view stems in part from a desire to simply get the system established, and able to answer a range of questions. But, in point of fact, a specific answer is probably the one a user would like, as opposed to being given a manual page to sift through, as we all have done.

Use the command **who** in order to see who's on the system right now.

The revised target binding set in displayed in Figure 6.15. It is the same in most regards as Figure 6.12b, with the exception that the roles of the clauses are reversed. *see who's on the system right now* is a subordinate clause of *use the command who* in Figure 6.15, and is bound as its **purp**. In Figure 6.12b, *use the command who* is a subordinate clause of *see who's on the system right now*. The subtrees representing the clauses are internally unchanged, although *user* remains bound as the **auth** of the core verb of whichever clause is the main clause. The semantic predicates "understand" that, unless a distinct **auth** binding is created for a subordinate clause of these kinds, the **auth** binding of the main clause verb may be transferred. To resolve the more important transformational problem, semantic predicates must also "understand" that a main clause that supports a **purp** clause must be the **means** to achieving the purpose. Therefore, if the bindings belonging to the **purp** clause match those of the **q**-set minus the reserved nonspecific binding with *how*, the main clause in the target is a match for the reserved binding.

---

user:1//noun(sg) >--- auth(n) ---> use:1//vb(cvb^base).
use:1//vb(cvb^base) >--- cvb ---> 0.
command:3//noun(sg) >--- obj(n) ---> use:1//vb(cvb^base).
who:4//pronoun(intrg) >--- instof(n) ---> command:3//noun(sg).
see:8//vb(cvb^base) >--- purp(v) ---> use:1//vb(cvb^base).
who:9//pronoun(intrg) >--- auth(n) ---> be:9//vb(cvb^base).
be:9//vb(cvb^base) >--- obj(v) ---> see:8//vb(cvb^base).
system:12//noun(sg) >--- advmod(loc^on) ---> be:9//vb(cvb^base).
right:13//adj(_) >--- advmod(s) ---> now:14//adv(_).
now:14//adv(_) >--- advmod(s) ---> be:9//vb(cvb^base).

Fig. 6.15 Target set for *[User] Use the command who in order to see who's on the system right now.*

Other inferential kinds of knowledge can rapidly rise in complexity. Only a limited amount of semantic predicates have been designed to cope with this problem. As a fairly simple example, another rephrasing of the question in Figure 6.12 will be considered:

How can I see who's working on the system right now?

The binding set generated for this question is identical to the q-set listed in Figure 6.12a, with the exception that the verb *work* replaces the verb *be* throughout. To effect the transformation of the revised q-set to the original target set in Figure 6.12b, the retriever must rely on semantic predicates that "know" that *working on the system* implies *is on the system.* It is relatively easy to code a hard-wired semantic predicate that supplies this piece of information, but the system is too small yet to be able to demonstrate the general purposefulness of this approach.

The primary reason that binding sets as a whole should form tree structures is shown by all the examples in the section, 6.12-6.15. Though subtrees may be composed of unequal numbers of bindings, a subtree in one binding set can only be matched with a subtree in another binding set since it is their equivalent case relationship to their respective parents that is the most important factor in the way that their information content relates to the whole idea. Semantic predicates do not attempt to match portions of two subtrees from a single binding set. It has already been pointed out that subtrees do not overlap, that is, binding sets as graphs are acyclic. It is this feature of the binding sets that allows the matching procedures of the retriever to work as they do. English is not as neat in this regard as this representation is, however, and two possible discrepancies need to be addressed.

The first discrepancy between English and this representation is the cases where an idea expressible as a single subtree in one binding set is split between two subtrees in another. This is a common enough occurrence that it cannot be ignored. However, the occurrences among the texts and surveys studied fall into two categories that can be approached in ways consistent with the representation. One type of exception is idioms.

These do not normally cause the double subtree problem. Idioms in general require special treatment in any representation scheme, and one piece of stored knowledge (in this case, a semantic predicate) must be devoted to each idiom. Semantic predicates for idioms handle the double subtree problem as well as any other exceptional relationships that arise from them.

The second type of exception to matching single subtrees can be characterized as resulting from a sort of ellipsis. Example 6.15 presents a simple case. By switching the hierarchy of clauses, the subordinate clause being elevated to main clause status, and vice versa, the subject *user* also switched parents, from the core verb of the original main clause, *see*, to the new main clause, *use*. The semantic predicates can still match new subordinate clause, without an explicit **auth** binding, to the **q**-set from Figure 6.12a, which has one, because it is clear that the **auth** binding of the main clause in Figure 6.15 is transferable to the subordinate clause. Double subtree problems, apart from those resulting from idioms, seem to be able to be solved through the principle of *transferability*. These are the cases where a modifier or other dependent *object* is stated only once in the language of the input, but by implication attach to other elements in the sentence.

What appears to be another discrepancy between English and this representation, particularly the way in which it is employed in the matching process, is actually not a problem at all. The retriever matches binding sets by recursively matching subtrees defined by the binding sets. The tree structure of Figure 6.2b has lost a simple tree structure since the node *file* is a dependent of two separate trees. The entire graph structure representing the subordinate clauses contains only that single point in common with the main clause however. Therefore, there is no ambiguity about the point in the matching process at which the subordinate clauses must be matched. By disregarding the direction in the binding arrows, this secondary graph may be regarded as a dependent of the node *file*. In other words, when the retriever comes to match the **obj** binding to *recover*, from the associated **q**-set (Figure 6.1) to a corresponding binding in a target, it should treat *file* and the subordinate clauses together as a single subtree. Within this subtree, the dependent subtrees may be matched recursively as

before.

CHAPTER VII

TOP-LEVEL VIEW OF THE COMPONENTS: A COMPLETE TRACE OF AN INPUT

A bottom-up approach to presenting UGURU was adopted in these pages. This was done because it was believed that, in an area of research where there is so much to be done and so few absolutes, an explanation of the principles that shaped the implementation is the primary concern. This explanation hopefully demonstrates the reasonableness and consistency of the design choices taken. Chapters IV and V described the nature of the grammar adopted and the elements of UGURU's parsing function. It included a brief explanation of the preprocessing function. Chapter VII described the retrieving function that takes as its input the output of the parser, and returns as its own output, an answer to a user question. It preceded this with a description of knowledge representation scheme. In this chapter, top-level code for the whole system, and of the components will be presented first, followed by a trace of an input through the system.

7.1. UGURU's Top-level Code

Figure 7.1 lists the top-level code for the system. The predicate **uguru** is the startup command for the system. A compiled version of this predicate, along with the compilation of

---

```
uguru :-
        promptuser( UserStmt ),
                /* this subunit prompts the user: the input may be statement or question.
                   Reading is performed by subpredicate rd, based on Clocksin/Mellish
                   text. The argument produced, UserStmt, is a [linked] list of lower-
                   case words.                                          */
        preproc( UserStmt, SysInput ),
                /* this subunit rewrites UserStmt according to numerous respelling and
                   rewording predicates. These handle contractions, certain idioms, com-
                   mon compound nouns and phrases. No misspellings or unknown words
                   are checked here.                                    */
        parse( SysInput ),
                /* this subunit parses the preprocessed user input. It checks unknown
                   words as it goes using subpredicate getTOKinfo. It traces the various
                   parse trees in parallel through its syntactic recognizers, applying
                   syntactic and semantic knowledge conjointly. It references two tables
                   of parsing rules: shift rules and reduce rules, chooses the best parse,
                   and asserts it as a binding set into the data base. The binding set is
                   marked as either a question (q-set) or statement (s-set).  */
        refine,
                /* this subunit's primary responsibility is to streamline the binding set
                   produced by the parser, mainly removing redundant modifiers, e.g.,
                   determiners, and other redundant expressions. If the binding set is
                   represents a statement, it is assigned a fact number and a fail call
                   is issued to return program execution to promptuser.  */

        respond,
                /* This subunit implements the retrieving function: it matches a q-set to
                   one of the numbered binding sets in the knowledge base. It calls var-
                   ious semantic predicates to do so: wordsame, edgesame, and insignif.
                   Then it outputs an answer, if possible.              */
        reinit.
                /* This subunit removes any temporary assertions in the data base, and
                   issues a fail call to return the system to promptuser.  */
```

Fig. 7.1 Top-level code for UGURU: correlation of the system components.

---

all UGURU's code, exists in a file called **ugusave**, which is automatically restored when Prolog is invoked to execute the program. In order to satisfy a call to **uguru**, Prolog successively evaluates each of the six subunit calls: **promptuser**, **preproc**, **parse**, **refine**, **respond**, and **reinit**. The names of the subunits fairly well identify their respective functions. **promptuser** and **preproc** gather and preprocess the user input; note that vocabulary look-up does not occur during these initial functions, but is reserved for the parser. The parsing function is implemented by the subunit, **parse**, which determines a single binding set to represent the input passed to it from **preproc**, and also by **refine**, which serves as a link between the parsing and retrieving functions by evaluating and simplifying the parser output, if possible. The retrieving function is implemented by the subunit, **respond**, and **reinit** reinitializes the system for other user inputs.

One feature of the code merits some clarification, especially for those who may be less familiar with the Prolog language. It is the lack of arguments in several of the subroutine calls. The values required by the subroutines have been asserted into Prolog's data base, where they are accessed essentially as global variables. For parameters that are structures comprising large amounts of information, it can be more efficient to use data base assertion and deassertion rather than accumulating these structures on Prolog's system stack. A second advantage in using assertion and deassertion is that this is the basis for a method of simulating parallelism in Prolog [Brat 86].

The second unusual feature is that no loop is apparent to return system to the **promptuser** subunit. **reinit** removes unwanted assertions into the system's data base, and issues a **fail** call, unstacking the calls from the several subroutines preceding it. This keeps the system from saving unnecessary values on the stack. Previous arguments of value to the system will have been stored by assertions into the data base, and are not destroyed by the **fail** predicate. There is a **fail** call hidden in the **refine** subunit as well. As it evaluates the parser output, which has been asserted into the data base, it checks whether the user statement was a question or statement. If a question, a q-set exists in the data base, and will subsequently be processed by the **respond** subunit. If a statement, the corresponding binding set

must simply be numbered and left in the knowledge base.[1] Since there is no processing to be done by the retriever, the **refine** subunit itself issues a **fail** call that restarts the system.

7.2. Invoking UGURU in Unix

The user entry into the UGURU program is a short shell script called "uguru" (see Figure 7.2a). The shell script pipes the startup predicate, also named **uguru** (see Fig. 7.1) into a system call to Prolog by reading it from a file called "gofile" (see Figure 7.2b). The shell script in file "uguru" then switches the source of input to a program called "reader." The Prolog interpreter has already begun to process the goal **uguru**, and has reached the subgoal **promptuser** which contains input request statements. The "reader" program is a filter that normally echoes the characters received from the terminal. If there were no further requirements from "reader," it could simply be replaced by the Unix command *cat* without arguments. However, one potential problem for UGURU is that interruption may abort the goal

---

[contents of file "uguru":]

```
( cat gofile ; reader )  |  prolog -s -q ugusave
```

Fig. 7.2a Unix shell script for invoking UGURU

[contents of file "gofile":]

```
:- uguru.
```

Fig. 7.2b Startup goal for initiating UGURU in the Prolog interpreter

---

[1]For the new fact to be saved for recall by UGURU at another time, the **refine** subunit must also store the binding set for the fact in a local file existing in the user's account.

uguru and leave the user directly in the Prolog interpreter. Therefore "reader" catches a break or interrupt signal, and sends a quit command to Prolog.

The necessary knowledge base and system predicates for UGURU are restored to the Prolog interpreter when it is invoked through the argument "ugusave." File "ugusave" is a save file that contains the complete state of the UGURU system. The file "loadfile" contains the names of several secondary loadfiles that identify all necessary locations of UGURU code for the Prolog interpreter. Instructing Prolog to load "loadfile" rebuilds the system which may then be saved in a file such as "ugusave." Restoration through a save file is very much faster than reloading the original code.

The remaining arguments in file "uguru," the *-s* and *-q*, are options modifying the performance of the interpreter. *-q*, for example, *quiets* the interpreter: normal output statements generated by the interpreter, such as its prompts and acknowledgements, are suppressed so that the user will only see I/O initiated by UGURU.

7.3. A System Trace

The remainder of Chapter VII traces the processing steps taken by UGURU to return an answer to the user question

How can I find out who is working on the system?

The knowledge base contains a fact, numbered 6, whose binding set expresses the Unix information:

To tell who is on the system now, the user should use the command *who.*

The following two sections divide the trace into two parts, mirroring the division of the system into its two major components, the parser and the retriever. Section 7.3.1 shows how the input is preprocessed, and then steps through the parsing cycles, following the actions of

the parser as it produces the binding set, called the **q**-set throughout this thesis, that characterizes the input. Since there are many recognizers active during each parsing cycle, only the recognizer that represents a step in the derivation of the correct parse tree will be examined. The section supplies the number of other recognizers active in each cycle during a sample run. As information continues to be added to the system, this number will vary.

Section 7.3.2 shows the series of transformations performed to equate the **q**-set and the target fact. The section demonstrates a correct sequence leading to the desired result. It does not show unsuccessful attempts that are eliminated by backtracking. The complete sequence taken by the retriever varies according to the order in which the semantic predicates are written, and also as the amount of information grows.

7.3.1. Trace of the Parser Actions

Preprocessing simplifies the original stream of tokens by replacing the phrase *How can I* by the single token *howcani* and *find out* by *find_out*. Subsequently the parser will see the input sequence:

howcani find_out who is working on the system ?

Since there are nine tokens in the input stream after preprocessing, there will be nine parsing cycles. It should be recognized in the following descriptions that, while the details of the genealogy of each recognizer are omitted, its parentage is always stored as its last parameter. In the sequence below, a parent would always occur in the previous cycle. Note also that the bindings are listed in a simplified form.

CYCLE 1:
New Input Token: howcani
Token Identification: special token, includes:
user:a3//noun(sg)
be_able:a2//vb(aux^base)
how:a1//adv(intrg)
All three tokens are implied by howcani. I is replaced by user.
Number of Syntactic Recognizers at the end of the cycle: 1
State of Recognizer leading to correct parse tree:

cls(main^qu(how))//[auth(n)]^0

This recognizer senses that a "how to" question is being parsed, and that the author of the action has been found. The main verb is yet to come.

New bindings created or completed:

how >--- means ---> ?
user >--- auth ---> ?
can >--- aux ---> ?

CYCLE 2:

New Input Token: find_out
Token Identification: find_out:2//vb(cvb^base)
Number of Syntactic Recognizers at the end of the cycle: 3
State of Recognizer leading to correct parse tree:

cls(main^qu(how))//[auth(n)]^actv

'find_out' was shifted into the previous recognizer as a verb phrase. Two reductions occur: the verb phrase is complete, and reduced as a verb object; the verb object is reduced as the main verb of the main clause. The difference between this recognizer and the previous one: the main verb has been found.

New bindings created or completed:

find_out >--- cvb ---> 0
user >--- auth ---> find_out
can >--- aux ---> find_out
how >--- means ---> find_out

CYCLE 3:

New Input Token: who
Token Identification: who:3//pronoun(intrg)
Number of Syntactic Recognizers at the end of the cycle: 9
State of Recognizer leading to correct parse tree:

cls(intrg^who)//[auth(n)]^0

'who' is shifted into the previous recognizer as part of a noun phrase beginning an interrogative clause. Two reductions complete the noun phrase as a noun object, and identify the noun object as the author of the main verb of the clause.

New bindings created or completed:

who >--- auth ---> ?

CYCLE 4:

New Input Token: is
Token Identification: be:4//vb(aux^base) and
be:4//vb(cvb^base)

'be' may be either the main verb or serve as an auxiliary verb: both options must be evaluated. Here it is an auxiliary.

Number of Syntactic Recognizers at the end of the cycle: 13
State of Recognizer leading to correct parse tree:

phr(vb)//vb

This recognizer senses that 'is' is an auxiliary verb (the first of the two options in its token identification) and is part of a verb phrase. Since the verb phrase is incomplete, no reductions occur.

New bindings created or completed:

be >--- aux ---> ?

CYCLE 5:

New Input Token: working
Token Identification: work:5//vb(cvb^prespcp)
Number of Syntactic Recognizers at the end of the cycle: 35

State of Recognizer leading to correct parse tree:

cls(intrg^who)//[auth(n)]^actv

After shifting 'working' into the previous recognizer as another part of the verb phrase, two reductions can occur. The verb phrase is now complete, and reduced to a verb object. Then the verb object can be reduced as the main verb of the interrogative clause. A semantic reduction also occurs that weights this interpretation of 'is working' much higher than the alternative, for which recognizers also exist, where the main verb is 'is' and the predicate object is the noun 'working' (e.g., 'happiness is working').

New bindings created or completed:

work >--- cvb ---> [clause]
be >--- aux ---> work
who >--- auth ---> work

CYCLE 6:

New Input Token: on
Token Identification: on:6//prep(_)
Number of Syntactic Recognizers at the end of the cycle: 28
State of Recognizer leading to correct parse tree:

phr(prep)//on

This recognizer expects to see a prepositional phrase that has been introduced by 'on.' It asserts that this prepositional phrase is being used as an adverbial object. (Other recognizers will assert that is used as an adjectival object.) No reductions are possible.

New bindings created or completed: *none*

CYCLE 7:

New Input Token: the
Token Identification: the:7//adj(det)
Number of Syntactic Recognizers at the end of the cycle: 28
State of Recognizer leading to correct parse tree:

phr(noun)//adj

This recognizer is processing a noun phrase. Only an adjective portion has been found, and therefore no reductions occur.

New bindings created or completed:

the >--- adj(det) ---> ?

CYCLE 8:

New Input Token: system
Token Identification: system:8//noun(sg)
Number of Syntactic Recognizers at the end of the cycle: 193
State of Recognizer leading to correct parse tree:

cls(main^qu(how))//[obj(v),auth(n)]^actv

'system' is shifted as a noun into a noun phrase. A number of reductions follow. First, the noun phrase can be completed as a noun object, which can be reduced as the object of the preposition 'on.' The prepositional phrase is completed and can be reduced: the information kept in the genealogy of the parent recognizers asserts that the prepositional phrase is an adverbial object within the interrogative clause. The clause has all necessary components and it too is reduced. It is the object of the main verb in the main clause, 'find_out.' All bindings that have been created are now in the single stack in the original recognizer generated for the main clause. A semantic reduction has also occurred, that boosted the weight of any recognizer asserting an attachment of 'work' and an adverbial prepositional phrase beginning with 'on.'

New bindings created or completed:

system >--- advmod(loc^on) ---> work
the >--- adj(det) ---> system
work >--- obj ---> find_out

CYCLE 9:

New Input Token: ?
Token Identification: ?:9//symb
Number of Syntactic Recognizers at the end of the cycle: 38
State of Recognizer leading to correct parse tree:

cls(main^qu(how))//end

Fourteen of the existing recognizers are syntactically legal sentences that can be completed by the final punctuation. Many are similar to the correct one, but the prepositional phrase 'on the system' is attached incorrectly. The correct one has the highest accumulated weight.

New bindings created or completed: *none*

7.3.2. Trace of the Retriever Actions

When the parser has finished, a **q**-set has been produced that represents the input question. The **refine** predicate eliminates trivial bindings, such as those involving determiners and some auxiliary verbs. When the **q**-set is handed to the retriever, it contains the following bindings. Throughout this section simplified versions of the bindings will be employed.

how >--- means ---> find_out
user >--- auth ---> find_out
find_out >--- cvb ---> 0
who >--- auth ---> work
work >--- obj ---> find_out
system >--- advmod(loc^on) ---> work

The retriever begins by looking for a goal in the **q**-set. In this case, it finds the binding from *how*, and proceeds on the assumption that this is a "how to" question. There are two kinds of facts in the knowledge base that can satisfy a "how to" question. One will contain a binding of the **means** case to a verb that is equivalent to the core verb in the **q**-set. The other is a fact that contains a binding of the **purp** (purpose) case from a verb that is equivalent to the core verb in the **q**-set. The retriever will look at all facts that contain one of these possible bindings, and explore further any that seem promising. Fact #6 in the knowledge base contains binding of the **purp** case from the verb *tell* to its core verb *use*. The retriever satisfies itself that *find_out* and *tell* can be equivalent in certain circumstances and proceeds to try to match the target set representing fact #6 to the **q**-set. The target set is as follows:

tell >--- purp ---> use
who >--- auth ---> be
be >--- obj ---> tell
system >--- advmod(loc^on) ---> be
now >--- advmod ---> be
user >--- auth ---> use
use >--- cvb ---> 0
command >--- obj ---> use
who >--- instof ---> command

As it begins its attempt to match the two binding sets, the retriever puts aside the binding with *how* in the **q**-set and the bindings in the target set that it assumes to be a match to the *how* binding and therefore an answer to the question. These include:

use >--- cvb ---> 0
command >--- obj ---> use
who >--- instof ---> command

The retriever identifies the target binding with *tell* as the main verb of the subtree that must match the tree dependent on *find_out*, and now proceeds to recursively match all subtrees.

Each successful match will be numbered as a separate step. As explained earlier, only a successful sequence will be traced.

(1) For the initial match following the entry to fact #6 these bindings remain unmatched.

q-set:
user >--- auth ---> find_out
who >--- auth ---> work
work >--- obj ---> find_out
system >--- advmod(loc^on) ---> work

target set:
who >--- auth ---> be
be >--- obj ---> tell
system >--- advmod(loc^on) ---> be
now >--- advmod ---> be
  [user >--- auth ---> use]
  [use >--- cvb ---> 0]

Every child in the tree with root *find_out* must be matched by a child in the tree with root *tell*. The retriever tries the first child in the q-set: the binding of *user* to *find_out*. A semantic predicate that understands implications tells the retriever that a verb of purpose, such as *tell*, without an explicit subject has the same auth binding as its parent. The retriever as allowed to manufacture the binding

user >--- auth ---> tell

This new binding matches exactly with the binding from *user* in the q-set. Since neither of these bindings have children of their own, i.e., there are no bindings of the form

? >--- ? ---> user

the retriever considers them matched and puts them aside.

(2) The remaining bindings to be matched are now:

q-set:
who >--- auth ---> work
work >--- obj ---> find_out
system >--- advmod(loc^on) ---> work

target set:
who >--- auth ---> be
be >--- obj ---> tell
system >--- advmod(loc^on) ---> be
now >--- advmod ---> be

Each binding set contains a single subtree now. The root of the one in the q-set is *work*, and the root of the other is *be*. The retriever invokes another semantic predicate that understands that "being on the system" is implied by "working on the system." It can pretend that it has replaced *work* with *be*. The subtrees will now match as long as all the children match. The first child is the binding from *who* to *be (work)* with the auth case. Since there is an exact replica in the target set, and both bindings are childless, the submatch is complete and the *who* bindings are put aside.

(3) The remaining bindings to be matched now include:

q-set:
system >--- advmod(loc^on) ---> be (work)

target set:
system >--- advmod(loc^on) ---> be
now >--- advmod ---> be

This step is easy, since there are again two identical childless children: the bindings from *system*. They can be put aside.

(4) There is only one binding left:

q-set:

target set:
now >--- advmod ---> be

The existence of an unmatched binding causes the whole attempt to fail, unless the single binding is known to be trivial or can be implied in the corresponding binding set. In this case, a semantic predicate allows the retriever to create a binding in the q-set from *now* to *be (work)* with the reasoning that, if one is on the system, he must be on the system now. The manufactured binding matches exactly with the one in the target set. Since both are childless bindings, they are both put aside.

The task of matching the q-set and target set is now complete. The retriever can refer to the first bindings set aside as equivalents for *how*, and generate an answer from them to the original question:

Use the command: who.

CHAPTER VIII

CONCLUSIONS

Two phases, mostly but not entirely distinct from one another, exist in the work done to create UGURU: design and implementation. Each phase has presented challenges of its own. Previous chapters, with the exception of Chapter VII, have primarily described the aspects of the system's design and the several earlier designs that eventually led to the current one. These chapters included examples from the code illustrating the specific representation of many of the design features as they appear in the implementation. Chapter VII covered the top level of the code itself, and exhibited a trace of an input through the system.

8.1. Current State of the Implementation

Once the design was worked out and implementation began, very few adjustments to the design itself were necessary. This should be considered a strength of the design. The adjustments that have been made are almost all changes in the number or nature of parame-

ters. These were necessary to maintain adequately complete information on the various objects existing within the system or to make access to the information contained in parameters more convenient or consistent. Of the problems encountered during coding of the system, nearly all involve adjustments to parameters. The unfortunate aspect of making these changes is that they had to be made everywhere in the code, and sometimes this involved some rather involved and tedious tracing.

On the whole, implementation progressed quite smoothly and straightforwardly. Again, this should be regarded as a great strength of the design. An important aspect of the design was to create a small working "prototypical" base that could be extended in a consistent way. The extensions so far, which mostly concern the addition of semantic knowledge and facts to the knowledge base, have not caused the unlearning of knowledge acquired earlier, nor exposed basic flaws in the parsing or retrieving systems. Extensions can frequently be made fairly easily, and when they are not, it is normally due to complexities in setting up all parameters properly and determining those that may be safely left uninstantiated. Learning in UGURU is now handled directly by adding code, which is always written in Prolog; the creation of a special language for specifying new information at a higher and more readable level from which the necessary Prolog code could be compiled would be an invaluable aid.

The system is still quite small. It contains approximately 6000 lines of Prolog code, not including the vocabulary and the facts in the knowledge base. This, too, is actually a strength, since these 6000 lines display the capability of handling a wide range of the linguistic and semantic problems involved in the project. The parser could be extended to handle the vast majority of normal syntactic sequences even in written English with the addition of a fairly small number of shift and reduce rules. However, the addition of code for semantic recognizers and semantic predicates, aimed at giving the system the specific knowledge for resolving the inevitable ambiguities in the user's language, will drive the size of the code up dramatically for some time as the system learns to answer more and more questions. A fundamental question about the viability of UGURU is, what happens as the number of semantic predicates continues to rise, and must they continue to rise at a rate that is linearly

proportional to the number of new questions that the system learns to answer? It is important that semantic predicates are "reusable," that is, eventually UGURU must parse and transform binding sets not previously encountered without the addition of new semantic knowledge being necessary.

At this time, that question cannot be answered conclusively. From the work done so far, there is the evidence that many semantic predicates do already play a part in producing answers for each of several distinct questions. That is, of course, a good sign, and it is expected that, with the coding of a large number of semantic predicates, the growth of the system would stabilize, but it cannot be projected at what point the need for large amounts of new semantic knowledge might taper off.

Of course, any reasonable natural language system requires a very substantial amount of semantic knowledge. This is true as well of the model, human beings, and cannot be expected to be avoided. A second question about the viability of UGURU arises when the bulk of the necessary code is considered, and how efficiently it is encoded, since the system is likely to run slower and slower as more information that might be relevant to a given input must be checked. This problem will be somewhat augmented by the addition of other shift and reduce rules that will further multiply the number of recognizers that are managed at any one point in time by the parser. Again, this is not a question that can be answered now, since the system is far from the size it would ultimately need to be, and the indications do not present a clear direction. In some cases, the time required to generate an answer has risen substantially as the amount of semantic knowledge has increased; in others, it has been affected only slightly.

The amount of time required to produce an answer varies. It is not dependent on the superficial appearance of the input question itself. Short questions may take much longer to answer than longer questions, for example. The fastest time on any question currently understood by the system is just under one minute, and the longest is over six minutes.

This would be unacceptable for a working system. These time lengths can be qualified on the basis of several factors, however. They are based, for example, on the fact that the parser performs a complete trace of all recognizers without the benefit of the **choose6** predicate, which eliminates unpromising parse trees from further consideration. **choose6** has not been generally activated within the code since complete traces are still considered valuable for building and understanding the system. In several cases, it was activated, however, and, on the average, cut the time by about fifty percent. Another qualifying factor is that the parser reports the sequence of steps that it takes, and therefore generates a substantial amount of I/O, sometimes as much as twenty printed pages. Surprisingly, the elimination of these output statements has reduced the times to produce answers only by slightly more than ten percent. Finally, it can also be pointed out that the times involved partially result from the environment in which the program runs, interpreted Prolog on a multi-user system. In a different environment, particularly on a parallel system that could take advantage of the parallelism in the design of the parser, execution times could possibly be cut dramatically.

While the questions brought forward above cannot be answered, based on the current state of UGURU's implementation, the initial goal of the implementation was to demonstrate that certain principles in the design have value as tools in natural language processing and may warrant further investigation. The relative ease of coding the core of the system, and incrementally adding semantic knowledge supports this point of view.

The appendix to this thesis contains a Unix script file that records a session with UGURU. In it, approximately forty questions are asked and answered, covering about twenty different facts from the knowledge base. Several questions referring to the same fact appear consecutively so that they can be easily compared. These questions exhibit some of the variations that the system can handle with regard to the content of the question, its syntactic organization, or the semantic knowledge necessary to evaluate it correctly. Questions were selected during this first stage of development to test and study what semantic knowledge must be added to the system to allow it to respond properly to each question. There is no reason not to assume that any other set of questions and facts that were used as a

basis for the first steps in development would not have produced results equivalent to those that were found using the set actually chosen.

In general, the results of incrementally adding semantic knowledge to support the ability to answer new questions has been promising. Adjustments to the basic parsing mechanism or the retriever's transformation scheme have not been necessary. Rewriting previously encoded semantic knowledge was occasioned only infrequently, and then only to qualify values that were originally without instantiated values and were activated in inappropriate contexts. Expressing semantic knowledge accurately and consistently within the formats already established in the code has been generally straightforward, the main difficulty being the correct treatment and inclusion of the required parameters. The only coding that remains for the system to grow is the addition of semantic predicates and recognizers, and a small amount of shift and reduce rules. All fundamental code is in place, assuming that conflicts do not develop as the semantic code increases.

8.1.1. Current Status of the Knowledge Base

UGURU's knowledge base currently contains twenty-six facts. They include information on the use of commands such as *who*, *passwd*, *mkdir*, *rm*, and *cp*. There is also information explaining the difference between *Mail* and *mail*, the definition of the term *pathname*, and also how to modify commands such as *cat* or *vi* with options. The facts fall into two categories, those answering "How do I ..." questions, and those answering "What is ..." The categories apply to content rather than phrasing. For example, the question

What is the command for printing a file?

is a "how to" question. The preprocessing function rewrites it to

How can I print a file?

even before the question is input to the parser. In a more complete knowledge base, there

would be information satisfying questions that ask "Where ...", "When ...", etc. Since different predicates had to be created for each nonspecific place marker,[1] the decision was made to concentrate on the two most frequent and important, "how to" and "what is" questions.

An example of the first category, which includes most of the twenty-six facts, is fact 9, a binding set representing the statement

The user can list the files in a directory by using the command *ls*.

Chapter VI described the process by which the retriever function matches a q-set representing a question to a target set representing a fact. In this case, any question whose binding set contains a nonspecific place marker, such as "how," attached as the **means** of the core verb, and whose remaining bindings match the target fact, minus those that comprise the phrase "by using the command *ls*," should retrieve this fact as its answer.

As noted above, the literal phrasing of the question may not indicate the type of question it is. It is also true that the facts can be expressed in several ways. The manner of expression should be transparent to the retriever, and it accomplishes this by the transformations it performs on binding sets. For this reason, "how to" facts were inserted into the knowledge base in about equal numbers in two formats. One corresponds to fact 9. An example of the other is fact 23:

In order to remove a file, the user should use the command *rm*.

No matter which format is used to express the fact, the retriever will still match it to an appropriate question. By examining the appendix, the reader cannot tell, and should not be able to tell, how the fact was originally stated.

---

[1]The nonspecific place markers are the interrogative words. They determine the type of binding that must be sought in the target binding set for the answer. See Section 6.3.

The few facts in the knowledge base that answer "what is" questions have only a single format, "[Subject] is [definition]." Alternative formats do not occur frequently enough to warrant implementation. An example of a "what is" fact is fact 17:

> An a.out file is a file containing the executable image of a program source file, produced by a compiler invoked by commands such as *pc* or *cc*.

8.1.2. Training the System and the Selection of Items

The questions that appear in the recorded session with UGURU listed in the appendix exemplify the types of questions that were chosen to train the system for answering. At this stage, as the system core is still being built, both the parsing function and the retrieving function require adjustments. These adjustments are the addition of new semantic information in the form of recognizers and predicates, and changes in the values of weights in the semantic recognizers.

Any new input to the system must be processed properly both by the parser and by the retriever. For the parser, new semantic knowledge may be required to select the right alternative among several legal syntactic choices, or provide an alternative when there are only syntactic dead ends. It may also be needed for recognizing idioms currently unfamiliar to the system.

Besides finding the targeted fact for a question phrased exactly as the fact is stated, the retriever must be able to match questions that are rephrasings of the fact. Each rephrasing may require new semantic knowledge for the matching process to succeed. Chapter VI discussed the transformations the retriever performs to effect the matching of corresponding binding sets. It accomplishes this through predicates that identify synonyms or equivalent phrases, discard nonessential or redundant words, reposition bindings to reshape the tree structures inherent in the binding sets, or add bindings that may be inferred from the context.

Training the system through increasing its semantic knowledge is not automated in any way, nor does the design include features that could be directly implemented to accomplish

it. Semantic knowledge must be added by hand, which means writing Prolog code for the purpose. A great plus for the system would be the creation of an interface that would allow new information to be specified in a more high level manner with automatic translation to Prolog code. Certainly there are many patterns that repeat among the semantic predicates, and programming them can become somewhat routine. Programming decisions regularly must be made about the degree of generality that a particular semantic predicate can assume: the more general it is, the more cases it can cover. However, greater generality increases the unpredictability of the effects of the predicate. The decisions so far seem to have worked well, and only minor changes were required. The greatest difficulty was handling the details of the parameters to the predicates, where the complexity invited some mishaps.

It is believed that as the system grows, the rate at which this semantic information must be added will peak and then decrease. Existing predicates will be used in many other situations than the question that occasioned them. Even with the relatively small number of questions and facts now known to the system, several new ones were answered without requiring new code.

The system design provides a second method for adding new facts to the knowledge base. The current binding sets representing facts have been created by hand and directly added as Prolog code. However, since the parser can produce binding sets from statements, these can be passed on to the retriever which will then store them within the knowledge base.[2] In this way, the system can learn facts from the user, who is able to specify them in English. The implementation of this part of the design cannot be fulfilled until a "talk" function also exists that can turn complete binding sets into English. The creation of the "talk" function presents difficult problems of its own, but the relative closeness of binding sets to English sentence structure should make the task more manageable. The "talk" function could be added to UGURU's core without changes being necessary to either the parsing or retriev-

[2]The knowledge base must be stored in at least two locations for its protection, however. The permanent core of facts must not be accessible to the user.

ing function.

8.2. Strengths and Weaknesses

The remainder of this chapter will be devoted to assessing UGURU's strengths and weaknesses, especially those pertaining to several key issues in natural language processing. The strengths and weaknesses of the implementation are, by and large, the strengths and weaknesses of the design, and so some theoretical aspects of the problem will be discussed. A synopsis of the current state of the implementation and some of its particular problems were presented in the introduction above. Among the issues to be covered below are UGURU's ability to acquire new information, that is, the nature of its extensibility; the algorithmic structure of the parser and retriever functions, that is, its use of parallelism and recursion, and how these relate to its efficiency; the nature of its knowledge representation scheme, and, in particular, the avoidance of a standardized primitive representation for a given "meaning;" and, finally, what design changes now seem to be warranted in a subsequent version of UGURU.

8.2.1. Extensibility: UGURU's Ability to Acquire New Information

UGURU's design outlines several ways in which the system can acquire new information. They are not all implemented currently to equal degrees, however. The primary way that UGURU learns at present is still through the direct insertion of Prolog code representing new information pertaining either to vocabulary, semantics, or syntax. To train the system to answer a new question, it is still almost always necessary to add some information, usually semantic. None of the updating of information is handled automatically, and another difficulty is that all new information must be rendered into Prolog by hand. An interface to perform this task would be an invaluable aid.

UGURU's facilities for gathering new information from a user are still weak, but two rudimentary functions to do so already exist. UGURU will query a user about words unk-

nown to it, attempting to ascertain the part of speech and a synonym. Words so learned will not ultimately be stored in the central system vocabulary, but saved in a local file that can be consulted along with the main vocabulary when that user invokes UGURU at a later time. This is, of course, too inflexible a way of learning new vocabulary, and, in fact, runs contrary to the way people increase their vocabulary. A procedure that learns the new word as part of a phrase should be considered more "natural," and is more like the **meansame** predicates belonging to the retriever rather than the synonym predicates of the parser. Another technique that could be pursued, based on elements already available in the system, is to make educated guesses regarding the meaning of the word. Since all parsing actions are weighted as to their likelihood, an examination of the most likely continuations of the parse trees prior to the unknown word could generate a "definition" internally.

The other facility for acquiring new information from the user is UGURU's ability to parse statements and treat them as new facts that can be asserted into the knowledge base. This capability is implemented. However, it cannot be demonstrated or used at this point since the corresponding *talk* function within the retriever is not implemented. Facts in the knowledge base currently are not "pure" binding sets, but hybrid sets in which an answer is represented by a character string. This allows output of the answer directly by simply printing the string. A *talk* function would be able to translate a binding set, or portion of a binding set, into English. For example, the binding set representing the fact *The user can see who's on the system now by using the command 'who'* includes the normal bindings for all words up to the word *using*, but the character string *the command 'who'* is treated as a single entity, the **obj** of the verb *use*. When a question is matched to this fact, the word *how* in the question corresponds to the phrase *by using the command 'who'* in the target. Both have the **means** relationship to the core verb. The retriever is currently written to seek the **obj** of *use* and print it directly. Ultimately, UGURU requires the *talk* function, and each word of the phrase *the command 'who'* would be parsed and bound just as all the others are.

Eventually, the weight scheme should have the means for automatic adjustment to reflect a reward for a correct parse and retrieval, and punishment for incorrect conclusions.

Whether there is a theory for weighting the actions of the parser, both syntactic and semantic, is far from clear. This arithmetic has been treated heuristically, and the results have been dependable and generally straightforward. As the system has been trained to answer more questions, it has also become clear that not just any numbers will work. Adjustments to the weights and to the way accumulated weights are formulated during processing has been necessary in the semantic predicates to accomplish the correct binding of prepositional phrases and other elements less tightly controlled by the syntax. The solutions so far have not been difficult, but the fact that the adjustments were necessary was taken as a reassurance that the weight system was indeed an integral part of the overall design.

The comments above are candid admissions about gaps and limitations in the system's ability to gather new information, as currently implemented. UGURU now learns only by the direct programming of the information. On the other hand, automatic learning has not had the highest priority. The point can be made that the mechanism for learning new vocabulary could be upgraded fairly easily, and the ability to learn new facts from the user, which can be utilized once a *talk* function exists to reconvert them to English, is a distinct plus.

A far more important consideration is that new information *can* be added to the system, regardless of how it learns it. Implementation and proof of the validity of the design is based on the notion that the fundamental processing predicates could be written as a core, and could be incremented with new knowledge without creating problems that would force the rewriting of existing code. This is the principle of prototyping common to the building of expert systems. In this regard, UGURU has so far been very successful. As it gains the knowledge necessary to answer new questions, it has not unlearned its earlier questions. The nature of the knowledge representation has also allowed all semantic knowledge to be coded clearly, and for the most part, simply and accurately.

The relative ease with which new knowledge has fit into the system allows the conclusion to be drawn that the knowledge base of facts and the accumulation of semantic knowledge can be extended widely. There is, of course, the possibility that as the system

grows, new information will begin to collide with older information, and the system become confused, perhaps only rarely or perhaps more generally. There is no indication now, however, that this condition will come to pass. UGURU seems to be very extensible.

8.2.2. Efficiency of the Algorithmic Structures

The fact that the parser follows all possible parse trees in parallel and independently of one another suggests an implementation on a parallel computer. The data base of semantic knowledge is equally accessible to all the recognizer units active during the parsing cycles and could be stored in a common memory. Implementation on a parallel computer would allow the parser to run as quickly as it would to process the single correct parse tree.

The retriever executes an exhaustive depth-first search to perform the necessary transformations that match a q-set with a target set. Since transformations can be made on either the q-set or the target set, both options will be attempted if necessary. More time can also be wasted since the same transformations may be tried many times simply with the order switched. Prolog naturally performs a depth-first search as it looks for a successful sequence to achieve a goal. For this reason, it is simplest to implement an algorithm that executes in this manner. When coding was begun on the retriever, it was not known how effective the transformation scheme would prove to be, and so the simplest option was chosen to test it. The retriever can be written in a manner similar to the parser, with independent units and a common semantic memory space. When this is done, the entire system can be made to run on a parallel computer.

Further gains in speed could be achieved by augmenting the preprocessing function. Interpreting common patterns as always having a single meaning means giving up some of the generality which was a major point of study in developing UGURU. Sometimes the patterns would be misinterpreted, but these rare occasions would be balanced by greater efficiency.

8.2.3. Knowledge Representation and Primitives

Possibly the most important question that one can ask about an NLP system is, how does it know when two expressions mean the same thing, or nearly the same? If there is one standardized internal representation for each possible meaning, the problem that remains is translating natural language inputs into the internal representation. Schwartz maintains [Schw 87], "Is is now a well-accepted principle that understanding in a computer means producing a canonical, language-independent representation of the meaning of an input." This is a major tenet of Schank's theory of conceptual dependency, in which the language-independent representation, however, is not actually language-independent, but based on primitives that are, in fact, just words.

The idea of primitives is attractive not only because it seems to delineate the problems of understanding more concretely, but also because the system may be expected to run more efficiently. Too much is lost, however. Schwartz offers an example of a conceptual dependency representation. In it, *read* is represented as *attend one's eyes to a book*. Only in a very narrow context is this not an unacceptably shallow understanding of *read*. Among major authorities, Schubert, Goebel and Cercone [Schu 79] take up the problem of translation to primitives, discussing in particular Schank's arguments for their use. They also conclude that the idea is unworkable for general language understanding.

For UGURU, the notion of a standardized internal representation was rejected. In its place, the scheme of semantic transformations was developed. It is believed that the work on UGURU so far demonstrates the viability of this alternative, even though it may be less efficient. The major consideration for this decision was that word-based primitive systems are psychologically unrealistic: people do not think that way. It is not unfair to note that no NLP system has advanced greatly towards general understanding of natural language, as Schwartz himself says (see Section 2.5). The fundamental reason may be that, for many of these systems, the internal representation is inappropriate since it is still based on words.

A word is a discrete unit standing for a network of concepts. The network is perhaps unbounded. Manipulations performed on words by computers may not have the power to imitate the manipulations performed on concepts. Amsler's study of the dictionary [Amsl 80] (see Section 2.3.1) showed that the number of words needed to form the definitions of all words can be reduced to a relatively small set of primitive words. However, these primitives are among the most general terms in English. Normally, building a system from primitives implies that the primitives have a large degree of independence and specificity.

Hardin's work [Dewd 87] (see Section 2.3.1) showed some of the dangers of translation by synonyms. The shift in meaning from the original takes effect very quickly, and probably cannot be controlled.

The idea that language can be reduced to a more efficient primitive set of words is contradicted by an argument based on evolutionary principles. It seems hard to believe that, as humans have evolved, language has remained uneconomical. Evolutionary processes tend to streamline and remove excess.

Some internal representation must exist. Alternatives to the word-based, synonym-based lexicons now common must be found. The associative memories of Rumelhart and McClelland [Rume 86] and Pollack and Waltz [Poll 86] appear promising.

Due to these considerations, UGURU's knowledge representation scheme stores input in terms of the words actually used. The primitives that do exist stand for the relationships between the words. UGURU decides that two phrases are the same by inspecting both the bindings and synonyms of the words involved. The decision may also be based on the context created in the rest of the input. Reliance on synonymic equivalence is not a solution to the problems discussed here, and UGURU can probably be expected to encounter the same limitations that primitive based systems do. One of the goals in developing UGURU was to demonstrate that there are alternatives that are at least equally good.

8.2.4. Summary

To conclude, it would be appropriate to enumerate the good and bad points of UGURU's design that emerged during the process of implementation. First, it must be acknowledged that an NLP system that attempts to tackle such a large domain of language problems and types of facts must be implemented with a large proportion of its knowledge before its validity can really be demonstrated. However, the ease with which the system grew incrementally to the size that it currently is speaks very well for the design. No important redesigning was necessitated by the addition of new modules.

Two lists follow. The first one itemizes special features of UGURU, some of which are original, which are the strong points of its design. The second represents drawbacks. It is believed that most of the drawbacks could be rewritten without the loss of the features in the first list.

STRENGTHS

(1) The approach to formulating an *object-oriented* grammar for English is original. It stresses that different mechanisms ought to be used to process high-level and low-level constituents. One of its advantages is a consistent treatment of conjunctions.

(2) The activities of the parser are distributed among independent units that can operate in parallel. The data base of knowledge that they access is common to all. The algorithm of the parser could eventually be implemented on a parallel machine; ultimately, the problem of understanding natural language and the size of the necessary knowledge base is so large that parallelism must be utilized.

(3) Semantic knowledge interacts with syntactic processing concurrently throughout parsing. The weight scheme allows the effects of semantic analysis to influence parsing in appropriate ways.

(4) Semantic recognizers are brought forth only when needed. That is, the search space is kept small.

(5) The binding sets used to represent knowledge have proved very flexible and powerful. Furthermore, they have been used in a completely consistent manner, without requiring patching for numerous special cases.

(6) The scheme of transformations used by the retriever has proved a very effective way of handling the problem of demonstrating that two phrases have an equivalent meaning. This is also original work.

(7) Knowledge representation is NOT based on a system of primitive values. This notion has been rejected as unrepresentative of the way people actually think.

(8) UGURU can learn facts directly from the user, who can input them in English.

WEAKNESSES

(9) The lexicon is synonym-based. Showing that two phrases have the same meaning is inherently inaccurate and runs the risk of chains of synonyms or the other syllogistic relations of words like **cmptof** ("HAS-A").

(10) The retriever is not implemented as a parallel algorithm, as is the parser. Before UGURU could be transported to a parallel machine, this revision would be required. Furthermore, the current version of the retriever executes a complete depth-first search of its predicates. This kind of mindless search is very time consumptive and, at the least, needs to be improved. Possibly a weight scheme is a solution.

(11) Semantic knowledge cannot be added except through writing Prolog code for the required predicates. Because of the complexity of the parameters involved, this process is error prone. The system would be much improved with an interface that translates semantic information supplied in an accessible interfact language to "low-level" Prolog code.

(12) The speed of the system is quite slow and expected to become even slower as the knowledge base accumulates. While there are justifiable factors that could be eliminated, such as simulation through the Prolog interpreter and extensive I/O, it is not apparent what improvements could be made without implementation on a parallel computer.

APPENDIX

```
Script started on Mon Feb 22 12:57:28 1988
% uguru

yes
| ?-                    HOW MAY I HELP YOU?
                    YOU MAY ASK A QUESTION, OR
                GIVE ME SOME INFORMATION IN A STATEMENT
                  THAT I WILL REMEMBER FOR YOU.
|: How can I make a copy of a file?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: How do I copy a file?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: How can I make copies of files?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: How can I duplicate a file?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: How do I copy one of my files?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: What is the command to copy files?
use the command:  cp  original_file  duplicate_file

              WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
                    (TYPE CONTROL-D TO EXIT)
|: What is an a.out file?
a file containing the executable image of a program source file,
   produced by a compiler such as  pc  or  cc
```

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do you read through a file?
use either of the commands: cat filename or more filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How could I see what is in a file?
use either of the commands: cat filename or more filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I look at the contents of a file?
use either of the commands: cat filename or more filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I see the contents of a directory?
use the command: ls

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How does one list the files in a directory?
use the command: ls

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do you list all the files in a directory?
use the command: ls

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I see what is in a directory?
use the command: ls

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: What is the command to find out which files are in a directory?
use the command: ls

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: Is there any way to move a file inside another file?
use the line-editing command: r nameoffiletobemoved
when editing the file into which it will be moved

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I tell when a file was last modified?
use the command: ll filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I find out the last time I changed a certain file?
use the command: ll filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do I know when I file was changed?
use the command: ll filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do we submit a program to the instructor?
use the command:
submit -v instructorlogin assignmentname programfilename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I submit a program to my instructor?
use the command:
submit -v instructorlogin assignmentname programfilename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I create a directory?
use the command: mkdir directoryname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I create a new directory?
use the command: mkdir directoryname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I make a subdirectory?
use the command: mkdir directoryname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do you make directories?
use the command: mkdir directoryname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How do you find out where a file is?
use the command: find. To locate the pathname to a file beginning with
the letters qprog, but the remaining ones are unknown, use
find ~ -name 'qprog*' -print

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: What is an absolute pathname?
use the command: mkdir directoryname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: What is the difference between mail and Mail?
that Mail offers more complete messages to the user and is an enhancement of the older mail, which is not recognized by System V except in the bourne shell. When in doubt, use Mail

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: What is the command to mail a message to another system?
use the command: Mail username@systemname

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: How can you send a message to a person on another terminal?
use either of the commands:
talk other_persons_login or
write other_persons_login

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: How can I mail a file?
use the redirection operator with the Mail command, e.g.,
Mail addressee < filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: How can I switch passwords?
use the command: passwd

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: How can I change my password?
use the command: passwd

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: What is the command to get a new password?
use the command: passwd

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: How does one permanently change from the bourne shell to the c-shell?
use the command: chsh yourloginname /bin/csh

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)

|: What do you do to remove a file?
use the command: rm filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I compile a pascal program?
use the command: pc filename

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I see who is on the system now?
use the command: who

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
|: How can I find out who is working on the system?
use the command: who

WOULD YOU LIKE TO ASK OR SAY ANYTHING ELSE?
(TYPE CONTROL-D TO EXIT)
%
script done on Mon Feb 22 14:40:35 1988

BIBLIOGRAPHY

[Amsl 80] AMSLER, R.A., *The Structure of the Merriam-Webster Pocket Dictionary*, The Univ. of Texas Press, Austin, Tx., 1980

[Bogu 83] BOGURAEV, B.K., "Recognizing Conjunctions within the ATN Framework," in *Automatic Natural Language Parsing*, K.S. Jones and Y.A. Wilks (eds.), Ellis Horwood Limited, Chichester, England, 1983, pp. 39-45

[Brat 86] BRATKO, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, Reading, Mass., 1986

[Bris 83] BRISCOE, E.J., "Determinism and its Implementation in PARSIFAL," in *Automatic Natural Language Parsing*, K.S. Jones and Y.A. Wilks (eds.), Ellis Horwood Limited, Chichester, England, 1983, pp. 61-68

[Brow 87] BROWN, R.J., "An Artificial Neural Network Experiment," in *Dr. Dobb's Journal*, 12, 4 (Apr. 1987), 16-27

[Cerc 77] CERCONE, N.J., "Morphological Analysis and Lexicon Design for Natural-Language Processing," in *Computers and the Humanities 11*, 4 (1977), 235-258

[Chom 57] CHOMSKY, N., *Syntactic Structures*, Mouton & Co., The Hague, 1957

[Chom 65] CHOMSKY, N., *Aspects of the Theory of Syntax*, MIT Press, Cambridge, Mass., 1965

[Dewd 87] DEWDNEY, A.K., "Computer Recreations," in *Scientific American*, 257, 2 (Aug. 1987), 108-111

[Fill 68] FILLMORE, C., "The Case for Case," in *Universals in Linguistic Theory*, E.E. Bach and R.T. Harms (eds.), Holt, Rinehart and Winston, N.Y., 1968, pp. 1-88

[Fodo 77] FODOR, J.D., *Semantics: Theories of Meaning in Generative Grammar*, Harvard University Press, Cambridge, Mass., 1977

[Gazd 83] GAZDAR, G., "NLs, CFLs, and CF-PSGs," in *Automatic Natural Language Parsing*, K.S. Jones and Y.A. Wilks (eds.), Ellis Horwood Limited, Chichester, England, 1983, pp. 81-93

[Harr 85] HARRIS, M.D., *Introduction to Natural Language Processing*, Reston Publishing Co., Reston, Va., 1985

[Katz 64] KATZ, J. and FODOR, J., "Structure of a Semantic Theory," in *The Structure of Language*, J. Fodor and J. Katz (eds.), Prentice Hall, Englewood Cliffs, N.J., 1964, pp. 479-518

[Kay 73] KAY, M., "The MIND System," in *Natural Language Processing*, R. Rustin (ed.), Algorithmics Press, N.Y., 1973

[Kell 86] KELLER, R., *Expert Systems Technology*, Yourdon Press, N.Y., 1986

[Leve 79] LEVESQUE, H. and MYLOPOULOS, J., "A Procedural Semantics for Semantic Networks," in *Associative Networks*, N. Findler (ed.), Academic Press, Orlando, Fla., 1979, pp. 93-120

[Marc 79] MARCUS, M., "A Theory of Syntactic Recognition for Natural Language," in *Artificial Intelligence: An MIT Perspective*, P.H. Winston and R.H. Brown (eds.), MIT Press, Cambridge, Mass., 1979, pp. 191-229

[Marc 80] MARCUS, M., *A Theory of Syntactic Recognition of Natural Language*, MIT Press, Cambridge, Mass., 1980

[Mill 87] MILLER, G.A. and GILDEA, P.M., "How Children Learn Words," in *Scientific American*, 257, 3 (Sept. 1987), 94-99

[Mins 85] MINSKY, M. *The Society of Mind*, Simon and Schuster, N.Y., 1985

[Ober 87] OBERMEIER, K.K., "Natural-Language Processing," in *Byte*, 12, 14 (Dec. 1987), 225-232

[Piag 52] PIAGET, J., *The Origins of Intelligence in the Child*, International Universities Press, N.Y., 1952

[Poll 86] POLLACK, J. and WALTZ, D., "Interpretation of Natural Language," in *Byte*, 11, 2 (Feb. 1986), 189-198

[Quil 68] QUILLIAN, M.R., "Semantic Memory," in *Semantic Information Processing*, M. Minsky (ed.), MIT Press, Cambridge, Mass., 1968

[Rest 84] RESTAK, R., *The Brain*, Bantam Books, N.Y., 1984

[Rose 62] ROSENBLATT, F., *Principles of Neurodynamics*, Spartan, N.Y., 1962

[Rume 73] RUMELHART, D.E. and NORMAN, D.A., "Active Semantic Networks as a Model of Human Memory," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, 1973, pp. 450-457

[Rume 86] RUMELHART, D., MCCLELLAND, J.L. and the PDP RESEARCH GROUP, *Parallel Distributed Processing*, 2 Vols., MIT Press, Cambridge, Mass., 1986

[Scha 72] SCHANK, R.C., "Conceptual Dependency: A Theory of Natural Language Understanding," in *Cognitive Science*, 3 (1972), 552-631

[Scha 81] SCHANK, R.C. and RIESBECK, C. (eds.), *Inside Computer Understanding: Five Programs Plus Miniatures*, Lawrence Erlbaum Associates, Hillsdale, N.J., 1981

[Schu 79] SCHUBERT, L.K., GOEBEL, R.G., and CERCONE, N.J., "Structure and Organization of a Semantic Net," in *Associative Networks*, N. Findler (ed.), Academic Press, Orlando, Fla., 1979, pp. 121-175

[Schw 87] SCHWARTZ, S., *Applied Natural Language Processing*, Petrocelli Books, Princeton, N.J., 1987

[Simm 68] SIMMONS, R.F., BURGER, J.F. and SCHWARCZ, R.M., "A Computational

Model of Verbal Understanding," in *AFIPS Conference Proceedings*, 33 (1968), pp. 441-456

[Smal 80] SMALL, S., "Word Expert Parsing: A Theory of Distributed Word-Based Natural Language Understanding," TR-954, Department of Computer Science, Univ. of Maryland, 1980

[Walt 85] WALTZ, D. and POLLACK, J., "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation," in *Cognitive Science*, 9 (June 1985), 51-74

[Wile 81] WILENSKY, R., "PAM," in *Inside Computer Understanding: Five Programs Plus Miniatures*, R. Schank and C. Riesbeck (eds.), Lawrence Erlbaum Associates, Hillsdale, N.J., 1981, pp. 136-179

[Wile 84] WILENSKY, R., ARENS, Y. and CHIN, D., "Talking to UNIX in English: An Overview of UC," in *Communications of the ACM*, 27, 6 (June 1984), 574-593

[Wilk 78] WILKS, Y.A., "Making Preferences More Active," in *Artificial Intelligence*, 11, 1978, 197-223

[Wins 84] WINSTON, P.H., *Artificial Intelligence*, 2nd ed., Addison-Wesley, Reading, Mass., 1984

[Wood 70] WOODS, W.A., "Transition Network Grammars for Natural Language Analysis," in *Communications of the ACM*, 13 (1970), 591-606

[Wood 80] WOODS, W.A., "Cascaded ATNs," in *Journal of the Association for Computational Linguistics*, 6 (1980), 1-12