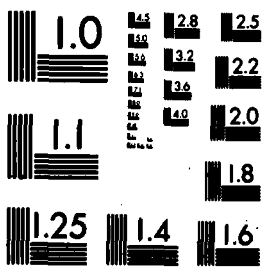
AD-A091 043   DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
A CASE STUDY IN NATURAL LANGUAGE PROCESSING: THE RUS SYSTEM.(U)
MAY 80   A W MANSKY   F49620-79-C-0131
UNCLASSIFIED   AFOSR-TR-80-1035   NL

1 of 2

1 OF 2

AD A 091043



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A091043

DDC FILE COPY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LEVEL II

(11)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 80-1035 ✓	2. GOVT ACCESSION NO. AD-A091 043
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CASE STUDY IN NATURAL LANGUAGE PROCESSING ✓ THE RUS SYSTEM	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Arthur William Mansky 411172	8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0131
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Delaware ✓ Department of Computer and Information Sciences Newark, DE 19711	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB Washington, DC 20332	12. REPORT DATE May, 1980
	13. NUMBER OF PAGES 121
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

RUS, semantic grammar, ATN, case grammar, semantically directed parsing, natural language parsers, ill-formed input

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The RUS system (Bobrow, 1978) is a new parser for English which allows semantically directed parsing by interleaving calls to a separate, independent semantic component with syntatic processing in a case-oriented ATN grammar. This thesis discusses the syntactic component (RUS), the points at which it calls a semantic component, and the nature of the syntactic-semantic interaction occurs, the nature of the interaction, and the structure of the dictionary are the only assumptions placed on the semantic component.

As a step toward building a user-oriented response facility to input that is not understood, we have developed diagnostic messages associated with the states of the RUS ATN grammar. A discussion of the general concepts involved is provided; the diagnostic message facility for each state is included in an appendix.

(18) AFOSR (19) TR- 80-1035

9 Master's thesis,

(6)

A Case Study in Natural Language Processing:

The RUS System*.

(11) May 80

by

(10) Arthur William Mansky

(16) 2304

(17) A2

(12) 133

Accession For	
NTIS GRA&I	☒
DDC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or special
A	

An abstract of a thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences.

*This research was sponsored in part by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Contract No. F49620-79-C-0131. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

(15) F49620-79-C-0131

May, 1980

Approved: ______________________

Ralph M. Weischedel, Ph.D.
Professor in charge of thesis

[illegible]

Abstract

A central design decision in developing a natural language processing system is its interfacing the syntactic and semantic components. Previous systems generally can be classed in one of two extremes in relating these two components. In the first extreme, syntax and semantics are totally separate. Although this allows the capturing of syntactic regularities and modifying of the two components separately, semantic "knowledge" cannot guide parsing. In the second extreme a "semantic grammar" includes both the syntactic and semantic knowledge about a particular application domain. Although such a system is very efficient when employed in the domain for which it was constructed, adding new constructions will not automatically include their syntactic paraphrases. Further, changing to a new domain usually requires a complete rewriting of the grammar.

The RUS system [Bobrow, 1978] falls between these two extremes. The syntactic and semantic components are separate, allowing independent modification of the components and capture of syntactic regularities, but they interact frequently during the parsing process, allowing semantic guidance of the parse. Only the semantic component need be changed for a new domain to be handled.

This thesis discusses the need for semantic as well as syntactic knowledge in a natural language processing system and the trade-offs involved in their interface design. The syntactic and semantic components of the RUS System are each examined separately; their representation and function within the system are outlined. The overall parser operation, with emphasis on the syntactic-semantic interface, is described. A user's manual gives the detailed steps needed to build the semantic component and experiment with the system.

As a step toward building a user-oriented error message facility, we have developed "meanings" for many of the states of the syntactic transition network. A discussion of the general concepts involved is followed by a few examples with explanation. All of the messages are included in an appendix. Another appendix includes the graph of the transition network.

By having frequent semantic interaction during the parsing, but keeping the syntactic and semantic components separate, the RUS System has the flexibility and efficiency that is necessary in a widely applicable natural language processing system.

A Case Study in Natural Language Processing:
The RUS System*

by

Arthur William Mansky

A thesis submitted to the faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences.

*This research was sponsored in part by the Air Force of Scientific Research, Air Force Systems Command, USAF, under Contract No. F49620-79-C-0131. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

May, 1980

A Case Study in Natural Language Processing:

The RUS System

by

Arthur William Mansky

Approved:____________________________________
Ralph M. Weischedel, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved:____________________________________
Hatem M. Khalil, Ph.D.
Chairman of the Department of Computer and Information Sciences

Approved:____________________________________
Richard B. Murray, Ph.D.
University Coordinator for Graduate Studies

Table of Contents

Chapter One

Introduction

When computers were first built, few people other than their designers were able to use them. As programming languages progressed from machine language to today's high level languages, more people were able to utilize the power of the computer. Taking this progression to its ultimate end, communication with the computer would be carried out as it is between people - via "natural language". That is, the language people use "naturally"; for us, of course, that is English.

The problem, then, is to build a system that allows the computer to emulate (to some acceptable degree) our own processing of "natural language" communication. Perhaps the system might be one to accept typed English input as a request to the system to perform certain actions. A database system allowing English queries is a typical example of this. While this may seem to be only a small subset of communication between people and computers, it is one that has been studied for many years, and has been found to be much more difficult a problem than many had anticipated.

The system would need to produce, from the English input, a form from which the intent of the user could be ascertained. This transformation from English input to a structured form is known as "parsing". For the system to "understand" natural language input, two types of knowledge are needed. First, the system must "understand" the rules of grammar that people use (usually subconsciously), the categorization of words, the extent of the vocabulary, and so on. This is syntactic knowledge. Second, the system must "understand" how meaning is represented within this structure, how context affects the interpretation, and so on. This is semantic knowledge. This marriage of syntax and semantics - the fields of linguistics and artificial intelligence - is known as natural language processing.

While the need to know the syntactic aspects of the input may be obvious, it is less clear that semantics plays a necessary role. A few examples illustrating the need for more than just syntactic knowledge follow.

The first example, from Chomsky, is one of a countless number of sentences which are grammatically perfect but semantically void.

"Colorless green ideas sleep furiously."

Purely syntactic analysis would accept the sentence, although it would be difficult to imagine any circumstance under which we would want it accepted. The remaining examples are not nonsensical sentences, but ones which require semantics to produce an interpretation nonetheless.

Unless we knew the whereabouts of each of the two men, the sentence

"The two men saw three boats tonight"

is ambiguous. We don't know the total number of boats seen. Clearly, if it is known that the two men were so distant from one another that a boat would not be able to travel from one's visual range into the other's in one evening, then we know that a total of six boats were sighted. Without any knowledge of the situation, we cannot know how many boats were seen.

Nominal compounds are another illustration of the need for knowledge about the "meanings" of words. Surely a construction entirely composed of nouns can hardly be analyzed in great depth using a purely syntactic approach.

"State motor vehicle inspection committee report"

and

"pressure cooker lid whistle adjustment screw" are two examples of this. Without knowledge of the inspection of vehicles, committees issuing reports, pressure cooker lids having whistles, and so on, "understanding" these compounds is hopeless.

Pronoun reference is another issue that is often unable to be resolved on syntactic grounds alone.

"I threw the big book on the chair and it broke."
"I threw the butterfly wing on the chair and it broke."

The "it" in each of the two sentences above, refers (most likely) to the chair in the first sentence, and to the butterfly wing in the second. Certainly the syntactic structure of the two was not the critical factor in assigning the referent of "it" (since their syntactic structures are virtually the same). Our knowledge of books, butterfly wings, and chairs allowed us to make our interpretation.

Paraphrase, i.e., using different syntactic structure, and often, different words, to communicate the same idea, is another case for semantic analysis in processing natural language. A purely syntactic analysis, for example, would find the following sentences differing in subject, prepositions used, and so on; while the meaning conveyed is virtually the same (except perhaps for which person is emphasized).

"Mary bought John a softball from Ted."

"Ted sold a softball to Mary for John."

There are many English sentences for which our general knowledge of "the world" is insufficient to allow us to choose only one interpretation. The context would (hopefully) allow us to determine which of the possible interpretations is most likely to be the one that was meant. One of the classic examples of this is the following sentence.

"I saw the man in the park with a telescope."

Among the possible interpretations are:

1. I saw the man who was in the park.
   I saw him by using a telescope.

2. I saw the man who was in a park.
   The park was the one with a telescope.

3. I saw the man who was in the park.
   He had a telescope.

4. I am cutting the man in half in the park.
   I am using a telescope to perform this deed.

While reading the above interpretations, the reader probably chose the interpretation he felt to be the most likely one (hopefully, not the fourth). This choosing of alternatives was not guided by the syntax of the sentence. Our knowledge of parks and telescopes help guide us. We quickly eliminate the fourth alternative (although it could be an acceptable sentence, in a very strange horror story, for example). We may feel that the first three are equally likely and would make a decision among those three based on the context in which the sentence appeared.

As a final illustration of the need for a sufficiently deep semantic component, the following sentence was input to a translator program, which translated it into Russian.

"The spirit is willing but the flesh is weak."

When the Russian was translated back into English, the following was obtained:

"The vodka is strong but the meat is rotten."

The processing of natural language by computer, then, demands that both the syntactic and semantic aspects of language are dealt with. The structures through which syntax and semantics are implemented in the RUS system [Bobrow, 1978] are discussed in the remaining chapters. Given that such structures exist, however, the question becomes: "How are these two components, syntax and semantics, to be combined within a single system?"

Generally, the goal is to express syntactic constraints in a general way while using semantic constraints to guide the parsing. The design of the RUS System is a synthesis of two approaches used in earlier natural language systems, the LUNAR System [Woods, et.al., 1972] and the SOPHIE System [Burton, 1976]. These two represent two extremes in the way syntax and semantics can be combined in a natural language processing system. The RUS System attempts to combine the best features of each while minimizing their shortcomings. First, we will briefly examine the two previous systems.

In the LUNAR System, the syntactic and semantic components are completely separate and each has its own knowledge representation. There are no interactions between them. In interpreting a sentence, the syntactic component first produces a parse using a grammar written in the Augmented Transition Network notation of Woods (discussed in Chapter Two). The resulting parse tree is then passed to the semantic interpreter which applies semantic rewrite rules to the tree and produces the final interpretation.

There are two advantages to such a system. First, that the syntactic and semantic components are distinct allows separate modification of the two. Only the parse tree format need be "standardized" as that is the only real interface between them. Second, the system has a substantial ability to capture syntactic regularities. Modifying the semantic component to allow a new sentence to be interpreted allows many syntactic variants of the sentence to be "understood" as well.

However, there are also serious disadvantages to this design. The most critical problem is the inefficiency arising from the inability of the semantic "knowledge" in the semantic component to guide the syntactic parsing process. The semantic interpreter can only accept or reject the syntactic parse tree. It cannot indicate how the parsing could be modified to produce an acceptable structure nor can it guide the parsing process itself in any way. Also, the semantic component has its own control structure to scan the parse trees in order to determine which rules are applicable. The complexity introduced by this "pattern matching" process further reduces the overall efficiency of the system.

At the other extreme, the SOPHIE System integrates the syntactic and semantic processes to such a degree that there no longer remains a distinction between the two. This "semantic grammar" approach is a set of grammar rules that, for each concept, characterizes all of the ways it can be expressed in terms of other constituent concepts. There is but one knowledge representation structure encompassing both syntactic and semantic analysis. Clearly, this representation allows semantic guidance of the parsing process, and therefore, is an extremely efficient methodology. In a system of this type, if the parser does anything at all, it produces an interpretation; i.e., there is no such thing as a sentence that is parsable but not interpretable.

However, because of this complete merger of syntax and semantics, when developing a semantic grammar for a new application domain, one often is not able to make use of the existing grammar. Even though the syntactic aspects of the new grammar might be similar to the old, because of the tightly merged nature of the representation, few of the similarities could be applied to this new domain as well. Also, expanding the grammar semantically to allow it to handle a larger semantic domain will not cause it to handle syntactic paraphrases in this new area unless they are explicitly included. The power of "generality" of a separate syntactic component is lost.

The RUS System is an attempt to gain the modularity and ease of modification of having separate syntactic and semantic components while allowing interaction between the two to a sufficient degree so as to gain the efficiency of having substantial semantic guidance of the parse. The structure that results is related to the "case structure" approach [Bruce, 1975] to language and makes use of a valuable property of English (and possibly all natural languages) referred to as "incremental parsability".

The RUS System, then, is able to capture syntactic regularities (by having separate syntactic and semantic components) and allow semantic guidance of the parsing process as well (by frequent semantic interaction from the syntactic component as it processes the input).

The syntactic component consists of an Augmented Transition Network (ATN) [Woods, 1973] that is compiled by Burton's grammar compiler [Burton, 1976] and is described in Chapter Two. The semantic component is discussed in Chapter Three, while Chapter Four explains the operation of the whole parsing process, including the interactions between the two components. A detailed examination of a portion of the system is presented in Chapter Five, revealing the "meanings" of many of the states in the ATN and illustrating an error message facility that would be invoked when a sentence fails to parse. Chapter Six details the user environment of RUS: how to construct semantic structures, interactive commands to aid the user of the system, and so on.

Chapter Two

Syntactic Component

The syntactic component of the RUS System is a compiled Augmented Transition Network (ATN) [Burton, 1976]. The ATN formalism has been used successfully in several natural language processing systems and has become the standard representation for natural language grammars. It has been argued that it is more efficient, powerful, flexible and perspicuous than other formalisms [Woods, 1973].

ATN's are based on the abstraction known as the Finite State Machine (FSM). A FSM is usually represented as a directed graph which has a finite number of nodes, each of which is labelled with a unique "state" name. The arcs between the nodes are labelled with symbols taken from a finite input alphabet. To process an input string, the FSM begins at a designated state (the "start state") and examines the first symbol of the string. A transition from <state x> to <state y> is made if and only if there is an arc from <state x> to <state y> whose label is the same symbol as is currently being examined. If such an arc exists, the transition to <state y> is taken and the input symbol is "consumed" (i.e., removed from the input string). The string is accepted by the FSM if the machine stops in one of its pre-designated "final" states and the entire input string has been consumed; the string is rejected otherwise. Finite State Machines accept the class of regular languages.

The ATN is a FSM that has been modified in two ways. The first modification is the addition of a recursion mechanism; this new formalism is known as a Recursive Transition Network (RTN). In the graphical representation, an arc may now also have a state name as its label. As in the FSM, if an arc is labelled with a symbol from the input alphabet, the transition is made if the "current" symbol is the same as the label. However, if an arc is labelled with a state name, the transition is made only if a substring (beginning with the "current" symbol) can be accepted by the RTN, where processing the substring begins at the state name given on the arc. That is, for the transition to be made, a "sub-machine" whose initial state is given by the arc label must accept a substring whose first symbol is the "current" one. The addition of this recursion mechanism brings the power of the transition network up to that of a pushdown store automaton. Recursive Transition Networks accept the class of context-free grammars.

The second modification made to the FSM formalism is the addition of registers. Now, in addition to symbols from the input alphabet and state names, arcs may also have any number of register setting operations and any predicate (which may examine the contents of any of the registers). A transition will now be taken only if the predicate evaluates to "true" and all of the conditions for a state transition in a RTN are satisfied. When a transition is taken, the register setting operations are performed. The registers may be used to hold any arbitrary list structure. Also, immediately before a recursive transition is made as described above, the current register values are saved and a new set of registers is created. The new registers are initially empty unless they are initialized to some value by the register setting operations on the arc that initiated the recursion. When the "sub-machine" accepts the substring, the value associated with acceptance of the substring is returned and the original register values are reinstated. The saving and restoring of the register values may be thought of as pushing the values on to and popping the values off of a stack. (In fact, the recursive call is made through a PUSH and the recursion ended via a POP, as is discussed below). The Augmented Transition Network, then, is a Finite State Machine with two additions, recursion (giving us a Recursive Transition Network) and register setting operations along with predicates. ATN's are equivalent to the class of automata known as Turing machines.

The Augmented Transition Network used in the RUS system syntactic component is based on the one described by Woods. As mentioned previously, to improve efficiency, the ATN is compiled and executed (as opposed to being interpreted). Rather than allowing any formulation of arcs, special arcs were designed to increase the ease with which ATN's may be written (and understood) by the user. The RUS ATN utilizes six "types" of arcs, each of which are described below.

```
(WRD <word or list of words>
           <condition><action>*(TO <state>))
```

In the WRD (word) arc, the first word of the input string is compared to the <word or list of words>. If there is just one <word> in the arc, the two are compared; if there is a <list of words>, the word from the input string is compared with each word in the list until a match is found or the list is exhausted. In either case, if there is a match, the <condition> is evaluated. If it is "true" (i.e., evaluates to a LISP non-NIL value), then the <action>* is performed. The star (*) in this notation is the Kleene star and indicates an arbitrary number (possibly zero) of items. Hence, <action>* represents zero or more actions. The actions may be register setting operations, calls to the semantic component (as discussed in Chapters Three and Four), or invocations of special actions which change the standard flow of control (as is discussed later in this chapter). Finally, the TO action causes the transition to be made to the specified <state> in the network.

(CAT <category><condition><action>*(TO <state>))

The CAT (category) arc is similar to the WRD arc except that the syntactic category of the input word (which is found in the word's entry in the dictionary) is compared with the category named by <category>.

(JUMP <state><condition><action>*)

The JUMP arc allows transition from one state to another without consumption of the input. If the <condition> evaluates to a non-NIL value, the <action>* is performed and a transition is made to state <state>. Since none of the input string is consumed, the state to which the transition is made may be thought of as a continuation of this state. In other words, the JUMP arc allows states to be structured in a way that may make the overall design of the ATN clearer, but it is not a necessary arc to have. Any ATN with JUMP arcs can be rewritten to an equivalent ATN without JUMP arcs. However, the state structuring freedom it allows is helpful in making the ATN more organized.

```
(PUSH <state><condition>(!<action>*)
                  <action>*(TO <state'>))
```

The PUSH arc is the recursive mechanism for the ATN. If the <condition> is true, a recursive transition is made to state <state>. The original register values are saved and a new set of registers are instantiated. The actions preceded by the exclamation point are "preactions"; they are performed before the transition is taken and typically assign values to the otherwise empty "new" registers at the lower level. When the substring is accepted by the subnetwork (if, indeed, it is) then the original register set is restored and the actions (other than the preactions) are performed, ending with the transition to state <state'>.

```
(POP <form><condition><action>*)
```

The POP arc indicates a final state. If the <condition> is true, the actions are performed and the <form> (a LISP expression) is evaluated. Its value is then returned to the PUSH arc that began the series of states that ended with this POP. The register values at this level are discarded and the original values reinstated before the return to the PUSH arc. The execution of a POP at the top level of the ATN indicates that the input string was accepted (i.e., we have a successful parse) as long as the entire input string has been consumed.

(VIR <constituent name><condition><actions>*(TO <state>))

The VIR (virtual) arc is used in conjunction with the HOLD action (described later in this chapter) to handle displaced constituents. If a constituent of the type <constituent name> has been placed on the hold list by a previous HOLD action, and the <condition> is true, then the constituent is removed from the hold list, the actions are performed, and the transition is taken to state <state>.

In addition to these six arc "types", the RUS ATN has a GROUP "super-arc". It has the following form.

(GROUP <arc 1>.....<arc n>)

The GROUP structure groups any number of arcs within any state together. As soon as the first state transition via one of the GROUPed arcs succeeds, none of the other arcs within the GROUP are tried. When arcs within a state are not GROUPed, the parser must provide back-up to that state. In many cases, though, there are states in the ATN that are not non-deterministic branch points. That is, there is at most one arc from that state that any given sentence can traverse. The GROUP arc allows the arcs in that state to be combined in order to signal the ATN Compiler that the state is actually a deterministic "choice-point", not a potential "branch-point". Any number of arcs within a state may be GROUPed.

The six arcs described above, along with the GROUP "super-arc", are the building blocks of the ATN. While the arc types and conditions constitute the primary flow of control mechanism in the ATN, the actions within the arcs actually build and manipulate the structures that the parser outputs. Some of the more common actions are described below.

The registers are the main mode of communication in the ATN. A state may set the values of certain registers to indicate what "knowledge" it has about the input; the values may then be examined at the next state to determine the flow of control from that point. Thus, an action that sets register values is among the most basic (and necessary) actions in an ATN.

(SETR <register> <form>)

The SETR (SET Register) action sets the register <register> to the value of the expression <form> (described below).

When a PUSH arc is taken, a new set of (empty) registers is created. The SENDR preaction allows the setting of a register at the lower PUSHed level from the present level.

(! (SENDR <register> <form>))

The SENDR (SEND Register) preaction is like SETR, except that the <register> is not set at the current level, but one level down, in order to initialize a register for the "sub-network" that was PUSHed.

(ADDR <register> <form>)

(ADDL <register> <form>)

The ADDR (ADD Right) and ADDL (ADD Left) actions modify the value of <register> by adding the value of <form> to the left or the right of the list that is the current value of <register>. The ADDL action may be thought of as a LISP CONS with the value of the <form> as the first argument and the current value of the <register> as the second argument.

(HOLD <form>)

The HOLD action is used in conjunction with the VIR arc to handle displaced constituents. This action places the value of the expression <form> on the hold list. As described earlier in this chapter, the VIR arc examines the hold list to see if there is a constituent of a particular type (NP, PP, etc.) being held.

Some of the more frequently used types of <form>s in the above actions (and in the POP arc) are described below.

(GETR <register>)

GETR (GET Register) returns the value (contents) of the register <register>. (SETR <reg 1> (GETR <reg 2>)) would set the contents of <reg 1> to be equal to the contents of <reg 2>.

The <form> "*" is essentially a special register that points to the current item being scanned. If the "*" appears in an action of a PUSH arc, or in a condition or an action of a VIR arc, then its value is the result of the embedded computation (if a PUSH) or the constituent being removed from the hold list (if a VIR). Otherwise, the "*" has as its value the root form of the currently scanned word of the input sentence. LEX is a <form> whose value is the current input word itself.

(BUILDQ <template> <argument list>)

BUILDQ is the basic data structure construction <form>. As the parse proceeds, the system records its findings in registers. These "findings" are LISP data structures of some complexity; BUILDQ is the primary data structure builder. Depending on the <template>, the elements of the <argument list> are treated in various ways, as shown below.

Template Element	Interpretation
#	Replace with the value of the corresponding member of the <argument list>
*	Replace with the "current" structure
+	Use the corresponding member of <argument list> (unevaluated) as a register name
!	Replace with an expression equivalent to the value of the <argument list> minus one outer layer of parentheses
!!	Same as ! above except that it modifies the <argument list> member

As an illustration of BUILDQ, if register HEAD has the value "ALUMINUM"; the variable CLASS has the value "NOUN"; the variable KIND has the value "(MINERAL)"; and the current input word is "SAMPLE", then

```
(BUILDQ (HEAD + CLASSIFICATION # MATERIAL
            ! # THING *) HEAD CLASS KIND)
```

would return the following form as its value:

```
(HEAD ALUMINUM CLASSIFICATION NOUN
     MATERIAL MINERAL THING SAMPLE).
```

In addition to the register-setting actions above, there are four actions which influence the flow of control of the ATN. The ABORT action causes the present configuration (parse path) to be stopped, and an alternative tried. The SUSPEND action provides a way to change the "weight" (likelihood of success) of the current configuration. This causes the configuration to be removed from active "status" and placed on the alternatives list. The most recent alternative with the best "weight" (possibly this same one) then becomes active. The RESUME action allows a lower level network that has already popped to resume processing input words from a later place in the input sentence (for handling extraposition). The RESUME simulates a PUSH to (recursive call of) the named state. The WAIT action is similar to the SUSPEND action except that the configuration "weights" are not changed. All alternatives are tried during the WAIT time. If no path was successful, then the "delayed" configuration continues.

In addition to all of the actions described above, calls to the semantic component are also among the possible actions in an arc. The next chapter describes the RUS semantic component; Chapter Four details the operation of the parser including how and where these semantic calls are used to guide the parse.

Chapter Three

Semantic Component

The semantic component of the RUS System is expressed in the form of a "case frame dictionary". The semantic "knowledge" in the dictionary is retrieved and processed through the use of a set of functions (described later in this chapter) that may be thought of as forming a "case frame interpreter".

The case structure concept has been successfully used in a number of natural language processing systems [Bruce, 1975]. The fundamental idea is that each word has a certain fixed number of relationships with other words; it can play only certain roles in a sentence. Which roles (or relationships) it fills depends on the context in which it appears.

The "case-oriented" semantic interpretation rules of the RUS System are associated with those words that can be HEADs of a syntactic unit (the main verb of a verb phrase, the main noun of a noun phrase, etc.). The rationale for defining rules for the HEADs of phrases is the way the overall syntactic/semantic parsing process operates. The semantic "checks" are made on a syntactic unit using the HEAD of the unit as the main element. Also, the cases of any word must be chosen from a list of available "case keys" [Mark, 1980]. These are syntactically determined keys, since the ATN must supply them. The parsing process, including the syntactic/semantic interaction, is discussed in detail in the following chapter.

The case frame for a word is placed on the word's property list under the CASESTRUC property. An individual case frame may be related to other case frames (i.e., semantic "knowledge" about other words) through an "inheritance" network. The form of the case frame determines the type of inheritance, if any.

Every case frame is either a list of clauses or a single atom. There are two types of clauses that can appear in the list-type case frame. The first kind of list-type clause defines a "case slot". This contains a "case key" (or list of "case keys") followed by a list of actions to be performed. The case key identifies the kind of slot. There are approximately thirty-five different slots for nouns, verbs, and prepositions. They are listed along with examples in the General Motors Report [Mark, 1980]. The actions to be performed are semantic tests, which check the validity of the syntactic constructs being proposed to fill the slot, and register assignments, which implement the semantic interpretation of filling the slot.

The second kind of list-type clause consists of a special atom (*V*, *PPCASES*, *PREMOD*, or *POSTMOD*) followed by the name of another dictionary entry. This indicates the sharing of case frames. It allows the current case frame to continue, using the cases of the entry named after the special atom. The special-atom/dictionary-entry statement may also be used as the last action within a clause. In that case, the current slot continues using the cases of the named entry.

If the whole case frame is just a single atom, then complete sharing is indicated. The atom must be the name of a different dictionary entry; it indicates that the case frame of this word is exactly the same as that of the named word.

These forms provide explicit linking between dictionary entries. Words in the application domain are "semantically similar" if they are connected in this inheritance network.

Another kind of inheritance in the dictionary is the SUPERC link. This allows the dictionary to be divided into a hierarchy of classes. If "pencil" has a SUPERC link to <writing-utensil>, then "pencil" is in the class

<writing-utensil>. (Words within the brackets are classes of objects. Their dictionary entries are of the exact same form as those for words.) In addition to defining the class relationships of words (or other classes), SUPERC can define the class relationships of phrase structures as well. SUPERC uses the FRAMETYPE of a phrase structure to represent it in the hierarchy of classes. FRAMETYPE is a semantic register that is set by the case frame of the HEAD of the phrase. This is an extremely powerful and useful facility, since it allows groups of words (not just single words) to participate in the inheritance network. Therefore, concepts that can only be expressed in more than one word can still be a part of the network. As there are many such concepts in English, this is an important feature in a knowledge representation scheme. Semantic "knowledge" about dictionary entries (words and concepts), then, can be shared partially, totally, or within a "subconcept-superconcept" relationship.

Each case frame is listed under one of six "properties" that are within the CASESTRUC.

PREMODCASES	(for nouns)
POSTMODCASES	(for nouns)
PPCASES	(for prepositions)
ADVERBCASES	(for adverbs)
DETERMINERCASES	(for determiners)
VERBCASES	(for verbs)

These "case types" are used by the semantic function calls within the ATN to specify which set of rules are to be used. The calls will also specify the slot type. These two pieces of information determine the set of actions that will be carried out.

A RUS-oriented lexicon was not available to us; however, another lexicon with a few simple "default" semantic rules allowed us to parse simple phrases. An examination of these CASESTRUCs will give the flavor of the notation; they are so basic as to be almost meaningless as examples of "semantic" rules.

```
((<NP>
    CASESTRUC
        (PREMODCASES ((HEAD (T (<= HEAD *))
                      ($ (T (E_- CASEKEY *))))
         POSTMODCASES (($ (T (E_- CASEKEY *))))
         DETERMINERCASES (($ (T (<= DETERMINER (BUILDQ
                                     (DETERMINER CASEKEY #
                                     DETERMINER *)
                                     CASEKEY)))))))
((<VP>
    CASESTRUC
        (VERBCASES (((HEAD SUBJECT OBJECT INDOBJ POP
                     NEG POSTRULES)
                     (T (E<= CASEKEY *)))
                     ($ (T (E_- CASEKEY *))))))
((<PP>
    CASESTRUC
        (PPCASES (($ (T (E<= CASEKEY *))))))
```

Three kinds of case slots are illustrated. A single case key-ed slot (the first slot of <NP> PREMODCASES), a multiply case key-ed slot (the first slot in <VP> VERBCASES), indicating that the rule is to be applied if the case key is among those listed, and a "catch-all" (the "$" slots) slot which is applicable regardless of the case key.

Following the case keys is the list of actions to be performed. If any in the list are unable to be performed (NIL in value), then no assignments are made using that particular set of actions. Another set of actions may be applied (if there is a "$" rule).

The first "action" in each of the above examples is the trivially true "T". The second action in each is an assignment of the structure that was given (to be put in a slot) to a "semantic register". The two registers HEAD and DETERMINER are explicitly named; in the others, the value of the case key that was requested by the syntactic component is the name of the register to be filled. Notice that the structure assignment functions begin with "E" when used with CASEKEY and do not when the register is explicitly named. The "E" indicates that the first argument (CASEKEY) is to be EVALuated to determine the name of the register to which the assignment is to be made.

The five "assignment operators" all assign a value (a list structure representing an interpretation) to a register. However, they differ in the way the structure is treated (listified, i.e., placed inside a pair of parentheses, or not) and in the way the register takes the assignment (replace or append) as shown below.

	structure to be assigned	register it is assigned to
<-	not listified	appended
_-	listified	appended
<==	not listified	replaced
<<==	not listified	replaced
<=	not listified	register is set to NIL if it was already assigned a value

No distinction in the way the "operators" <== and <<== make the assignment was found by me. The "assignment operators" _-, <=, and <== have "E" equivalents (E_-, E<=, and E<==) that EVALuate the first argument. This allows a structure to be assigned to a register (usually given by the value of CASEKEY) without the name of the register being explicitly written in the CASESTRUC rule.

The semantic "knowledge" in the system, then, is represented by the case frames as described above. The "case frame interpreter" is a set of functions that retrieve and process this semantic "knowledge." The top level functions that assign a semantic interpretation to a phrase are called from the ATN. (Where this is done is discussed in the following chapter.) All of these functions are similar to the function ASSIGN.

```
(ASSIGN
   <component> <casehead> <key> <casetype>
               <notassignflg> <syntaxcasekey>)
```

<component>	the phrase to be put in a slot
<casehead>	the head of the phrase
<key>	the CASEKEY
<casetype>	one of the six properties: POSTMODCASES, VERBCASES, etc.
<notassignflg>	if non-NIL, no register assignments are made

<syntaxcasekey> a more general CASEKEY; used with <key> to make a "non-standard" CASEKEY

When the ASSIGN function is called, the case frame of <casehead> is searched for the CASESTRUC property <casetype>. If the property is found, the list of slots is retrieved. The slots are then searched (by examining the case keys) to see if one is of the type <key>. If the appropriate slot is found, the tests and register assignments are executed. If any of them cannot be successfully completed (return non-NIL), then the assignment fails at that slot and all of the actions that were successful are undone; however, processing of the frame continues as there may be another acceptable slot to try. If all of the actions are successfully completed, ASSIGN returns a value of "T" (true) and the register assignments are made; <component> has been successfully placed in a slot, i.e., it has been "understood".

If the above process fails because of the absence of an applicable slot, then the HEAD word cannot have the suggested relationship with any phrase. If it fails because all of the actions in an applicable slot could not be successfully completed, then the HEAD word cannot have the suggested relationship with the particular phrase <component>.

Generally, the presence or absence of a slot and the tests (within the actions) of a particular slot embody the semantic "knowledge" of the application domain. The semantic checking of the RUS System is totally concentrated into the case assignment process. To move to a new application area, then, the user must define the words in the new domain by choosing slots, defining semantic tests to determine which concepts can fill a slot, and constructing register assignments that reflect the filling of the slot by the concept, i.e., that the concept was "understood".

The ASSIGN and ASSIGNQ functions are the invoking functions of the "case frame interpreter". The remaining portion of this chapter describes the case structure access and manipulation functions which the two use to assign a "meaning" to a phrase. They are usually called only by ASSIGN and ASSIGNQ and their sub-functions, not from within the ATN.

The functions ASSIGN and ASSIGNQ differ only slightly. ASSIGN calls the function GETKEY which returns the list of actions in the requested case structure slot. ASSIGN then takes this result along with its <component>, <key>, and <notassignflg> parameters and invokes the function ASSIGNCASEbyKEY. (In other words, GETKEY retrieves the appropriate case slot and ASSIGNCASEbyKEY "executes" it with the given component.) ASSIGNQ also does the above; and, in addition, places the value of <component> in the register SYN/SUBJECT if the <key> is SUBJECT or in the register SYN/OBJECT if the <key> is OBJECT. If the <key> is neither of the above, then ASSIGNQ is identical in action to ASSIGN, except that the third argument of ASSIGNQ is QUOTEd, i.e., not EVALuated.

The function GETKEY retrieves the list of actions from the appropriate case slot by using the functions GetPATH, GetINHERITEDCASE, and FINDKEY. Each of these are discussed below.

The function GetPATH takes a list as its argument. The CAR (first element) of the list is EVALuated; the function GetPATH1 is then called with the EVALuated CAR as the <gramitem> and the CDR (remaining elements) of the list as the <searchlist>.

```
(GetPATH1 <gramitem> <searchlist>)
```

GetPATH1 can be used to return the value of a particular property of a structure. This means that <gramitem> (perhaps the contents of a particular register) is searched for the value of its <searchlist> property. For example, if <gramitem> is `(HEAD NOUN ALUMINUM LEX ALUMINUM NUMBER SG) and <searchlist> is `NUMBER, then the call to GetPATH1 will return "SG".

GetPATH1 can also be used to find the Nth element of a list. This means that <gramitem> is searched to find its <searchlist>th element. For example, if <gramitem> is the value of a register such as PREMODS, which may hold any number of premodifiers of the head noun, and <searchlist> is 1, then the call to GetPATH1 will return the list that represents the first premodifier.

GetPATH1 can also be used to return the value of a word's CASESTRUC property. This means that if <gramitem> is a <word> and the <searchlist> is (: <property>), then the value returned is the value of the <property> property under the CASESTRUC of the word <word>. If the searchlist is (:E <property>) then <property> is EVALuated first to get the property name. For example, if <gramitem> is 'SHOW and <searchlist> is '(: PREMODCASES), then GetPATH1 will return the PREMODCASE CASESTRUC for SHOW.

Any (or all) of the three ways of using GetPATH1 described above can be combined by placing the <searchlist> elements together into a "super" <searchlist>. Beginning with the first <searchlist> element, the value determined by the element is found (as described above) and is used by the next <searchlist> element. This left-to-right evaluation continues until the last <searchlist> element is used. For example, if <gramitem> is 'ALUMINUM and the <searchlist> is '(: PREMODCASES 2), then GetPATH1 will return the second slot in the PREMODCASE CASESTRUC for ALUMINUM.

In GETKEY, GetPATH is used to "reduce" a register value to its HEAD word and to find the word's generic class (<NP>,<VP>,etc.) if necessary.

The function GetINHERITEDCASE is also used in GETKEY.

(GetINHERITEDCASE <frame> <case>)

GetINHERITEDCASE returns NIL if there is no CASESTRUC for the word <frame> or if the CASESTRUC has no rules under the property <case>. Otherwise, it returns the entire "rule set" listed under the property <case> of <frame>'s CASESTRUC. For example, (GetINHERITEDCASE 'SYSTEM 'PREMODCASES) will return the PREMODCASES CASESTRUC for the word SYSTEM or NIL if there is not a PREMODCASES CASESTRUC or no CASESTRUC at all.

The function FINDKEY is also used in GETKEY.

(FINDKEY <key> <searchlist>)

FINDKEY returns the list of actions under the <key> case key from the <searchlist> "rule set". The <key> is the case key being searched for and <searchlist> is the value of one of the caseframe properties (POSTMODCASES, etc.). The <searchlist> is the result of a call to GetINHERITEDCASE described above. FINDKEY looks for the appropriate "action list" in the following way: while there is still a rule to be examined, its first "element" is checked. If it is one of the special indirection atoms (*PREMOD*, *V*, etc.), then the new set of rules is retrieved by calling GetINHERITEDCASE for the word named after the special atom. Otherwise, if the first "element" of the rule is equal to "$" (the special "any-case" symbol) or the given <key> or if the <key> is among those listed in the first "element" of the rule, then the rest of the rule is returned as the value of FINDKEY. If none of the above are true, then the process is repeated for the next rule. This process is repeated until an applicable rule is found, or the "rule set" is exhausted and a NIL is returned.

GETKEY uses GetPATH, GetINHERITEDCASE, and FINDKEY to retrieve the appropriate "action list" in the following way.

```
(GETKEY <key> <word> <keytype>)
```

It first calls GetPATH as many times as necessary to get to the single word that is the head of the structure <word>. Then, a call to GetINHERITEDCASE and FINDKEY using <key> and <keytype> (<keytype> to specify the CASESTRUC subproperty to GetINHERITEDCASE) will result in the "action list" requested or NIL. If the result is NIL, the process is repeated with a call to GetPATH to find the generic <keytype> definition of the word <word>. Using the current lexicon, this means, for example, that the PREMODCASES CASESTRUC for <NP> will be used instead of the one for ALUMINUM, because one for ALUMINUM has not been written. That is, the invoking of GETKEY via

```
(GETKEY `HEAD `ALUMINUM PREMODCASES)
```

will cause the following:

1. Since the <word> is not a list, GetPATH is not called.

2. GetINHERITEDCASE is called and returns NIL, since there is no CASESTRUC for ALUMINUM.

3. FINDKEY uses the result from GetINHERITEDCASE to find the appropriate "action list", but since that result was NIL, this result is NIL.

4. Because the first call to FINDKEY was NIL, the two functions are called again, below.

5. GetINHERITEDCASE is called, but this time, using the generic <NP> instead of the word ALUMINUM. (This was found via GetPATH.) The PREMODCASES of <NP> are

returned as the result.

6. FINDKEY uses this result, searching through it for an "action list" under the HEAD case key. It is found and returned as the value of FINDKEY and, therefore, the value of the call to GETKEY itself.

GETKEY, then, is the function that retrieves the appropriate "action list". The ASSIGN (or ASSIGNQ) function can now call ASSIGNCASEbyKEY to try to make the "semantic interpretation" assignment.

```
(ASSIGNCASEbyKEY <casekey> * <caseframe>
   <noassignflg> <leftaddflg> <resetusedregs>)
```

The arguments to ASSIGNCASEbyKEY have the following meanings:

<casekey>	the <key> from ASSIGN
*	the <component> from ASSIGN
<caseframe>	the "action list" returned from the call to GETKEY
<noassignflg>	the <notassignflg> from ASSIGN
<leftaddflg>	(not used by ASSIGN)
<resetusedregs>	(not used by ASSIGN)

To summarize the progress of the assignment action up to this point, the appropriate "action list" has been found. We now want to evaluate the list, hoping that all tests and actions are successful so that the assigning of an "interpretation" to the <component> under this HEAD word can be made.

ASSIGNCASEbyKEY proceeds by examining the first "element" of the "action list". If this is one of the special indirection atoms (*PREMOD*, *V*, etc.) then the <caseframe> becomes the one for the word named after the special atom. In other words, the "action list" is now the appropriate one for the named word. (This is done by calls to FINDKEY and GetINHERITEDCASE.) The procedure is re-started (by using the CLISP "while") so as to allow any number of indirections before a real "action list" is found. Once one is found, a "T" is returned only if one of two conditions are true:

1. All of the actions evaluate to non-NIL. (This is the usual successful situation) or

2. The ACCEPT/IT/FLG is "T" (it is "T" if not changed), and the component to be assigned (*) is a list, and either its FRAMETYPE is <DUMMY> or <ELLIPSIS> or it has more than one HEAD sublist and the first HEAD is a PRONOUN.

Otherwise, the value returned by ASSIGNCASEbyKEY to ASSIGN (or ASSIGNQ) is NIL.

The functions described above are the primary functions of the semantic component. Some of the other functions (most of which use these) are briefly described below.

ASSIGNDET uses GetPATH and ASSIGNQ to construct the total determiner structure in DET and check it semantically to be sure that it can be the determiner with the HEAD noun.

ASSIGNVERBCONSTITUENTS takes a list of adverbs, question-adverbs, adjuncts, and other constructs and then calls ASSIGN with the HEAD verb (and VERBCASES) for each one.

BuildDET builds the structure that represents the determiner by combining the articles, possessives, quantities, ordinals, superlatives, simple determiners, and negative determiners.

BuildPP builds the structure that represents the prepositional phrase. It uses ASSIGNs to be sure that the head preposition, the prepositional object, and the preposition within the phrase (if any) all check semantically.

InterpBNP interprets the base noun phrase by calling ASSIGN for each of the premodifying phrases.

There are also fifty-six GRAMTESTFNS (grammar test functions) that are used in the conditions of the ATN. These functions perform "lookahead", i.e., check to see if certain syntactic constructs (words or phrases) could be next, and check the features of the words that the ATN encounters. These grammar test functions add a great deal of power to the condition checking on the arcs of the ATN.

In summary, the semantic component of the RUS System consists of special data structures that represent the semantic "knowledge" about words in the application domain (the "case frames"), and a group of functions that analyze this "knowledge" to produce the register assignments that represent the "understanding" of the input (the "case frame interpreter").

Chapter Four

Parser Operation

The two preceding chapters described the syntactic and semantic components of the RUS System in detail. Each component was explained while avoiding a discussion of the other as much as possible. The RUS System parses a sentence, of course, using information from both components. This chapter describes the overall parsing process: how the ATN deals with alternative paths, the RUS parsing "strategy", features to improve efficiency, the lexical prepass, the interactions between the two components, and the final output of a successful parse.

The ATN grammar compiler produces code that traverses the ATN using a depth-first control structure [Burton, 1976]. At any given state, the arcs are tried one at a time in the order that they appear there. The first arc that succeeds is taken and an alternative "configuration" (which will try the remaining arcs) is pushed onto the alternatives stack. If a "configuration" blocks (i.e., none of the arcs leaving the state succeed), then the top "configuration" on the alternatives stack is initiated. (The "normal" control structure of the ATN as described above can be altered through the use of the special actions ABORT, RESUME, WAIT and SUSPEND as discussed in Chapter Two.) A "configuration" is a data structure that completely characterizes the state of the processing. A typical "configuration" might contain the following:

Configuration Number	unique to this configuration; used to identify it in paths, an alternative lists, etc.
State	identifies the state in the grammar being examined and also indicates which arc within that state is under consideration.
Node	a pointer to the input.
Stack	a pointer to higher levels of the ATN that PUSHed to the present level.
Regs	a pointer to the list of registers available to this configuration.
Feats	a pointer to a list of feature registers.
Hold	the hold list of constituents that are still unassigned.

When transitions are made from state to state, only a few registers are usually changed. Each configuration must "know" the values of all of the registers; but storing all of the register values with each configuration would take up a great deal of memory. To handle this, the register-name/register-value pairs are stored on a "forked stack". This is a merged list data structure where the register values common to a group of configurations are stored once, and those specific to a particular configuration are stored with a pointer to the "common" list. Thus, only the pointer to the beginning of a configuration's register list is needed to obtain a complete record of the current register contents there.

The goal of the whole parsing process, of course, is to produce a representation of the interpretation of a phrase by representing the interpretations of its syntactic units and the relationships between them. The question is: "Can the interpretation of a phrase be built up from interpretations of its 'parts' as the parsing proceeds, or must all of the 'sub-interpretations' be made before an overall interpretation can be?" That is, either syntax or semantics (or both) may be "wholistic" rather than "incremental" processes. The results obtained in the RUS System indicate that English is, to a large extent, able to be parsed (and interpreted) in an incremental, left-to-right way. There is a large class of syntactic relations which can be determined incrementally; and these are sufficient to provide the necessary information to semantics. That is, as the syntactic role of a constituent in the total unit is determined, the semantic interpretation is constructed. The two processes are closely linked, not only because the semantic component uses the results of the syntactic component incrementally, but also because the ability of the syntactic component to assign syntactic roles without considering the roles of all other constituents is based on its ability to "check with semantics" to see whether or not a possible syntactic assignment "makes sense". There are some phenomena in English that cannot be handled in a strictly incremental way (such as extraposition), but they are able to be handled by simple extensions to the incremental scheme (VIR arcs and HOLD actions).

Four major features have been added to the basic ATN structure in an effort to increase the efficiency by making the parsing process as deterministic as possible. The amount of non-determinism is reflected (partially) in the amount of back-up that occurs wile parsing a sentence. The basic ATN structure is a non-deterministic mechanism; each arc traversal is thought of as a "branch-point" - the parser must be able to back-up and try alternative paths. A truly deterministic parser regards each state as a "choice-point", where a (correct) choice must be made, without allowing back-up to that state.

The first added feature is the GROUP arc. As discussed in Chapter Two, this allows the grouping of arcs within any state into a deterministic "super-arc".

The second added feature is "look-ahead". At many points in a typical ATN, there is the possibility of a certain type of constituent that will, therefore, be PUSHed for. That this constituent type is expected is usually based only on the structure that has been found so far. However, such a PUSH should be avoided if the current word (or the next few words) precludes that type of constituent. For example, even though a noun phrase might be here ("hypothesis-driven" approach), we would not PUSH for it if the next word clearly precludes its presence ("data-driven" approach); e.g., a noun phrase would not begin with a preposition or conjunction. It was found that looking no further than three words ahead is sufficient to prevent "obviously" incorrect PUSHes; in fact, often the next word is sufficient. Although there are cases where the three word look-ahead is not sufficient, they are relatively rare [Bobrow, 1978].

The third added feature involves a change to the depth-first control structure of the ATN. If we PUSH for a constituent and the result is semantically unacceptable, we must still allow for the possibility that a semantically acceptable constituent exists but that the first one discovered by the PUSH is not it. Because of the control structure of the ATN, all alternative possible constituents of the desired type will be found before any alternatives to the PUSH are tried. This assumes that it is more likely that there is a constituent of the desired type here which will fit into the current phrase than that the first semantically acceptable constituent of that type will fit somewhere other than in the current phrase.

However, as the parser becomes more deterministic, the latter is more likely to be true; i.e., the first semantically meaningful result returned from a PUSH is likely to be the best description of what occurs there. As an illustration, the parser may be analyzing the sentence "LIST THE ALUMINUM SAMPLES ON THE UPPER HALF OF THE SCREEN". While the parser is processing the noun phrase that begins with "THE ALUMINUM SAMPLES", it will hypothesize a prepositional phrase, and find "ON THE UPPER HALF OF THE SCREEN". Although this is the correct prepositional phrase to find at this point, it is not one that modifies "THE ALUMINUM SAMPLES". Under the normal ATN control structure, all other possibilities for prepositional phrases will be explored. Many useless parses may be generated before it is eventually decided that the noun phrase has no prepositional phrase modifiers and that the prepositional phrase actually modifies the clause. To avoid this, the control structure of the parser was changed so that the parser will produce only the first semantically acceptable result of a PUSH. If that is rejected as not "making sense" as a constituent of the current phrase, other branch-points which would produce alternative results for the PUSH are not tried until all other alternatives have been tried. In the above example, rather than trying to interpret the prepositional phrase differently, other alternatives (such as the end of the noun phrase) are tried. Now, however, it would be duplicating work to make the higher level (clause) network reinterpret the prepositional phrase that was already examined by the lower level (noun phrase) network.

The fourth added feature allows some of the work performed by the first attempt to parse a sentence to be re-used by the later attempts. The Well-Formed Substring facility (WFS) holds a popped constituent along with other information to ensure that the new PUSH that wants to use the result of an earlier PUSH has the same context.

These last two features allow a syntactically correct structure that was hypothesized at the wrong level of the network to be saved. Then, if the higher level PUSHes for the same type of constituent at the same place in the input, the previous result is found without further parsing. This eliminates the re-analyzing of a phrase that would otherwise result.

When a sentence is submitted to the RUS System to be parsed, it is first analyzed by a function that makes sure that a dictionary definition of each word in the sentence is in core. This "lexical prepass" expands abbreviations, combines words into compounds, performs a morphological analysis to produce canonical versions of words, and finds grammatical categories and features of words. Some examples of mappings that the lexical prepass performs are shown below.

```
          HIS ------> HE'
        , AND ------> AND
UNITED STATES ------> UNITED-STATES
       THINGS ------> THING (PLURAL)
```

The prepass categorizes each word of the input into lexical categories such as: N (noun), V (verb), ORD (ordinal), ADV (adverb), NPR (proper noun), and so on. If the exact word from the input is not found in the dictionary, it must have a root form that is in the dictionary. The root-finding procedure is as follows:

(1) if the word is a number or string, it is automatically "in the dictionary" as a generic; otherwise,

(2) look up the word in the dictionary; if not found,

(3) reduce the word to a root form by morphological analysis; look up this form; if not found,

(4) try punctuation analysis to isolate a root form; look up this form, if not found,

(5) print the message:

"I don't know the word:" <word>

"Please type its definition or correct spelling."

Once the prepass has been successfully completed, the ATN begins at the state given by the value of STARTSTATE, TOP/. At any given point, the network being processed defines "expectations" of what will be in the input at that point. In the RUS System, the most important expectation is the HEAD of the syntactic unit being looked for (the main verb in a verb phrase, the main preposition in a prepositional phrase, etc.). The way that elements of the sentence are assigned to the structure of their syntactic unit is defined by the case frame (as described in Chapter Three) of the HEAD word of the unit. The basic approach of the parser, then, is: look for the HEAD of the phrase (saving the non-HEAD words); find the HEAD; build the phrase structure defined by the HEAD; check the case frame of the HEAD to be sure that the elements found so far are acceptable; look for any other words that are needed to complete the phrase structure; look for the next HEAD. Parsing a sentence in the RUS System is this continual process of "suggestion" by the ATN and "criticism" by the case frame interpreter.

As an illustration of how the calls to the semantic interpreter are used to assign interpretations as well as guide the parse, the following is taken from the actions of the PUSH NP/ arc in the state VP/OBJ.

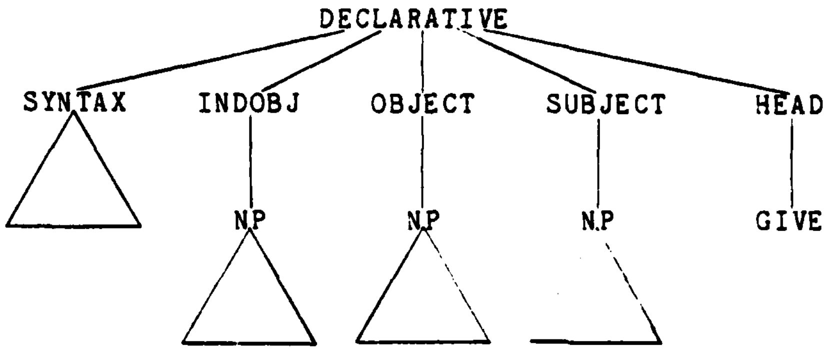
```
     (OR (AND (ASSIGNQ * HEAD OBJECT VERBCASES)
              (SETR OBJECT *)
              (ASSIGNQ OBJ? HEAD INDOBJ VERBCASES)
              (SETR INDOBJ OBJ?))
         (ABORT))
```

At this point in the parse, the verb phrase has been found (with HEAD holding the main verb), the noun phrase that was found and placed in OBJ? is believed to be the indirect object, and the noun phrase that was just successully PUSHed for (and is, therefore, the value of "*",) is believed to be the direct object. The two calls to ASSIGNQ allow the interpretation of the object and indirect object with the verb in HEAD to be checked. If both are successful, then the registers OBJECT and INDOBJ are set. Because the four actions are within a LISP AND, if all are successful, then the LISP OR has its first component satisfied (equal to LISP non-NIL) and the OR construct is completed. If any one is not successful, then the value of the AND is false and the second component of the OR is evaluated. This causes an ABORTion of the current configuration - either the proposed object or the proposed indirect object failed to be "understood" with the main verb in HEAD.

The parsing process proceeds until a parse is found or all paths have been exhausted. In the latter case, rather than output "NO PARSE", an error message facility is suggested as an addition to the RUS System and is described in the following chapter. If the parse was successful, the "value" of the parse is the value of the variable "**".

The output of the parse is printed from the value of the variable "**". (How to change what is displayed of ** is described in Chapter Six.) "Double star" is a list data structure that contains the values assigned to the registers during the parse. The word "SYNTAX" marks the beginning of a new level in the parse. At the end of each level, the value of the HEAD of that phrase is listed. To illustrate, the sentence "I GAVE YOU THE BOAT" was parsed (with only the "default" semantic rules as described in Chapter Three).

```
(DECLARATIVE SYNTAX
  ((NOADJCOMPLEMENT)
   (ASPECT (TENSEMARK TENSE ((TNS PAST))))
   (HEAD . GIVE)
   (STYPE . DECLARATIVE))
  INDOBJ
   (NP SYNTAX
     (HEAD PRONOUN HEAD YOU LEX YOU)
       DETERMINER
     (DETERMINER CASEKEY NIL DETERMINER NIL)
       HEAD
     (PRONOUN HEAD YOU LEX YOU)
  OBJECT
   (NP SYNTAX
     ((HEAD NOUN HEAD BOAT LEX BOAT NUMBER SG)
      (DET? DETERMINER ART
         (ART HEAD THE SINGULAR T PLURAL T))
      (ART ART HEAD THE SINGULAR T PLURAL T))
       DETERMINER
     (DETERMINER CASEKEY THE DETERMINER
        (DETERMINER ART
           (ART HEAD THE SINGULAR T PLURAL T)))
      HEAD
    (NOUN HEAD BOAT LEX BOAT NUMBER SG))
SUBJECT
  (NP SYNTAX
    (HEAD PRONOUN HEAD I LEX I)
      DETERMINER
    (DETERMINER CASEKEY NIL DETERMINER NIL)
      HEAD
    (PRONOUN HEAD I LEX I))
HEAD GIVE)
```



Although the above is a greatly abbreviated version of the actual output, it illustrates the representations of the levels. The result is a DECLARATIVE with HEAD equal to "GIVE" (in the past tense). The SYNTAX list of the phrase consists of some of its attributes; there is no adjective complement, the head is "GIVE" in the past tense, etc. There are three "sub-levels" in the phrase. The INDOBJ (indirect object) level is a noun phrase whose HEAD is "YOU", with no determiner. The OBJECT level represents the object of the sentence, "THE BOAT". Following SYNTAX, the values of the registers HEAD, DET?, and ART are listed. This noun phrase has a determiner of "THE" (which can be with a singular or plural noun) and "BOAT" as its HEAD. Finally, the third level is the

SUBJECT of the phrase. Its HEAD is "I"; it has no determiner. The "double star", then, contains the output of the parse in this level/sub-level representation, with SYNTAX signalling the beginning of a level and HEAD marking the word that is the head of the level and that the level is completed.

As the parse proceeds, each "sub-network" builds the syntactic unit that was PUSHed for (noun phrase, prepositional phrase, etc.), using the HEAD of the unit as a guide for what "kind" of words are expected to complete it; and, the "parts" of the unit may be checked semantically as well as the entire unit itself. Upon completion of a successful parse, the variable ** contains the list structures that are held by the registers as representing the "meaning" of the total input phrase.

Chapter Five
Meanings of States

A good error message facility is an essential part of any system that is purported to be "user-oriented". When using a high-level programming language, for example, we expect the compiler to output meaningful error messages. Any programmer who has encountered messages such as "ERR 4A76:35" is well aware of the importance of the content of these messages; and, certainly, a compiler that does not output any messages (except "PROGRAM COMPILED" or "PROGRAM NOT COMPILED") is totally unacceptable. Because of the relatively limited grammar of a programming language, a good error message mechanism for its compiler can be written with a reasonable amount of effort; in fact, the programmer expects such a facility.

In a natural language processing system, there is a much broader range of inputs that need to be processed. The grammar of a programming language is constructed so that it is unambiguous, and makes use of a limited "vocabulary"; but a natural language grammar is taken from actual human communication - with ambiguity and dependence on context. The user of a natural language processing system can exceed the "knowledge" of the system in a number of ways, each of which should be handled in a way that informs the user (as much as possible) why the system was unable to process the input. The RUS System does not currently have such an error message facility; the remainder of this chapter describes the technique we suggest for implementing one.

The simplest case in which the input can fail to be parsed is when it contains a word unknown to the system. In the RUS System, the lexical prepass (described in the previous chapter) will try to find, in the dictionary, the root form of each word of the input. If it is unable to find the root form of any word, it will ask the user to enter the correct spelling of the word or its definition.

More often, however, the user exceeds the system's "knowledge" of natural language in a more serious way. The user may input a phrase whose syntactic structure is beyond the range of the current grammar. This could be due to the existence of a class of acceptable syntactic structures that the grammar does not handle; or, the input was not (as much of our verbal communication is not) syntactically correct. Whatever the case, the error facility should explain, in as much detail as possible, why the input failed to parse. The input sentence may be within the "syntactic capability" of the ATN, but could contain a phrase that cannot be "understood" by the semantic analysis; the lexicon CASESTRUC rules for the words involved do not allow the semantic functions to succeed in assigning a "meaning" to the phrase. We are usually able to tell the user what component of the phrase was uninterpretable (subject, determiner, etc.), but the present system has no "reasoning" component to examine the CASESTRUC rules in order to form an explanation of what input could have been understood or why the present phrase could not be. In this case, the message we suggest at present is of a more general nature. Through the error message mechanism described below, we can determine which semantic check failed, but we are unable to explain, in a detailed way, why it failed.

As the input is processed, the ATN traverses various arcs, moving from state to state. Each state may be thought of as a representation of what has been understood, what is now expected, and where processing is to continue if what is expected is found. If the parsing process "blocks" at a state, this information can be used to explain to the user how his input did not meet the parser's "expectations". The "meaning" of a state, then, provides the necessary information for the constructing of error messages to be output if the parse blocks at that state.

The meaning-of-a-state/error-message representation that is used is that of Weischedel and Black [1979] and Black [1979]. The first problem is finding the state that best represents the greatest degree to which the parser "understood" the input. Since the parser will try many alternatives before they are all found to be unsuccessful, there are many paths, and therefore, many blocked states which could be considered. First, all of the paths which did not consume the greatest number of words are eliminated. Then, of the remaining paths, we want to select one path. The heuristic used is to determine the "length" of each path. This is equal to the number of arcs traversed not counting "trivial" PUSH or JUMP arcs (i.e., those with T conditions). The path with the greatest "length" is chosen as being the most likely one. If more than one path meets these criteria, then one is chosen nondeterministically.

Although this error message facility is not yet implemented for the RUS System, the "meanings" of many of the states have been written. When the error mechanism has been constructed, the parser will either successfully parse the input sentence, or it will invoke the error function for the state as described above.

The "meaning" of a state is represented as a series of condition/action pairs that are placed on the property list of that state.

```
(<state-name>
        (<condition 1> <action>*)
        (<condition 2> <action>*)
                 .
                 .
                 .
        (<condition n> <action>*))
```

The control structure is similar to that of the LISP COND. That is, <condition 1> is EVALuated. If the value is true (LISP non-NIL), then the <action>* (series of actions) is EVALuated. If the value of <condition 1> is NIL, then <condition 2> is EVALuated. This continues until a <condition> is non-NIL and its <action>* performed or the pairs are exhausted (all <condition>s are NIL and no messages are printed).

A <condition> may be any LISP predicate. In error messages for the states of the RUS ATN, the typical <condition> might examine the contents of a register, invoke a parser utility function to determine what may be following the current word, or call a semantic assignment function to reveal whether or nor such an assignment was successful at this point.

An <action> may be one of the following:

```
(PCHAR (QUOTE " <text> "))
        prints the characters in <text>

(PRINT-REG (GETR <reg>))
        prints the contents of register <reg>

(PRINT-ANY-STRING <spec>)
```

prints the value of <spec>;
<spec> may be a LISP expression such as (CADR (GETR HEAD)), or, more frequently, the symbol *, whose value is the root form of current input word or, if in a PUSH arc, the expression popped, or LEX, whose value is the current input word itself.

(TERPRI)
prints a carriage return/line feed

(EXAMPLES? <word>)
prints example phrases using the word <word>; the examples are on a separate disk file. Each word has a list whose first element is the word itself and whose remaining elements are the examples.

(CONDACT <state>)
transfers control to the condition/action pairs of the state <state>

(LOOKAHEAD)
causes the simulation of all transitions to other states from this one via arcs with unconditionally True conditions. The new state's condition/action pairs are then evaluated.

(; <comment>)
a NULL action. Allows any action, typically the first of a group of actions, to be a <comment>.

Two additional features are a macro facility and a message handling mechanism for embedded sentences. In defining a macro, the form of the condition/action pairs is the same, except the macro name replaces the state name. Then, any condition/action pair in any state (including those in another macro) may be simply a macro name. The messages for embedded sentences allow the printing of a message explaining that the higher level sentence was acceptable, so that it is clear that the error lies in the embedded sentence. The messages for the states that begin "looking" for an embedded sentence are written in the same way except that the state name is preceded by *BAKSTK*. This allows the separation of messages for blocking at a particular state and those for blocking while processing an embedded sentence from that state.

The error message facility will output the input sentence up to the point where the parse blocked. The printing functions above (PRINT-REG and PRINT-ANY-STRING) examine the form of their arguments and determine how to print them in a neat format. Because of the structure of the RUS ATN, and because there was not a RUS-oriented lexicon available, the EXAMPLES? and LOOKAHEAD functions were not used in writing the sample messages.

To illustrate how the condition/action pairs represent the "meaning" of a state, an example is explained below. The "meanings" of all of the states in the S, VP, and NP subgraphs appear in Appendix B.

```
 1 (ASPECT/TO
 2     ((AND (CAT V)
 3           (NOT (CHECKF V UNTENSED)))
 4      (; ASPECT VERB followed by 'TO' should then be
 5                             followed by an untensed verb.
 6                             This one is tensed.
 7      (PCHAR (QUOTE " AN 'ASPECT' VERB, SUCH AS ' "))
 8      (PRINT-REG (GETR HEAD))
 9      (PCHAR (QUOTE " ', FOLLOWED BY 'TO', EXPECTS AN
                                              UNTENSED "))
10      (PCHAR (QUOTE " VERB TO BE NEXT, AS IN: "))
11      (TERPRI)
12      (PRINT-REG (GETR HEAD))
13      (PCHAR (QUOTE " TO LEAVE... "))
14      (TERPRI)
15      (PRINT-ANY-STRING LEX)
16      (PCHAR (QUOTE " IS NOT UNTENSED. "))
17      (TERPRI))
18     ((NOT (CAT V))
19      (; As above, except we don't even have a verb.)
20      (PCHAR (QUOTE " AN 'ASPECT' VERB FOLLOWED BY
                                                 'TO' "))
21      (PCHAR (QUOTE " EXPECTS AN UNTENSED VERB, AS
                                                     IN: "))
22      (TERPRI)
23      (PCHAR (QUOTE "          IT BEGAN TO 'GROW'. "))
24      (TERPRI)
25      (PRINT-ANY-STRING LEX)
26      (PCHAR (QUOTE " IS NOT A VERB. "))
27      (TERPRI)))
```

At the beginning of the VP (Verb Phrase) group of arcs, an "aspect" verb, such as 'START', 'BEGIN', 'TRY', may be followed by the word 'TO'. If this is the case, the transition is made to state ASPECT/TO. We now expect an untensed verb. Blocking at this state can be due to having a verb that is not untensed (or, at least, is not marked in the lexicon as being untensed), or because the current word is not even a verb.

The condition/action pairs for state ASPECT/TO are shown above. Line 1 contains the name of the state.

Lines 2 and 3 form the first condition. Following the comment (lines 4, 5, and 6) are the printing commands of the message. To illustrate the effect of the commands, if the head verb were 'START' and the input word were 'SWAM' ('START TO SWAM'), the message that would be printed would be:

```
AN 'ASPECT' VERB, SUCH AS 'START'
, FOLLOWED BY 'TO', EXPECTS AN
UNTENSED VERB TO BE NEXT, AS IN:

          START TO LEAVE...

SWAM IS NOT UNTENSED.
```

If the word after 'TO' is not a verb at all, then the condition at line 18 would be true, and the second message (lines 20 thru 27) would be printed. If the input were 'START TO DESK', the message would be:

```
AN 'ASPECT' VERB FOLLOWED BY 'TO'
EXPECTS AN UNTENSED VERB, AS IN:

        IT BEGAN TO 'GROW'.

DESK IS NOT A VERB.
```

The condition/action pairs may use the values of the registers both to determine which message should be printed and as part of the output of the message itself. Because the information about a word is stored in the lexicon, the messages can guide the system builder in developing the lexicon. If a message seems to be "wrong", it may be because the information stored in the lexicon about a word in the sentence is incorrect and/or incomplete.

Because of the ability of the parser to "look ahead" using the parser utility functions, many state transitions are made knowing that the destination state will be able to process the input and, therefore, the parse cannot block there. Also, some state transitions are made if the input word is a member of one of a number of categories, and at the destination state, each of these categories has an arc. Again, unless there are semantic ABORTs, the parse cannot block at the destination state. Therefore, there are a number of states in the RUS ATN for which no error messages were written, since it is impossible for the parse to block at those states. The "meaning" of those states are explained in paragraph form at the end of Appendix B.

An example of such a situation is the Q/HOW state. The Q/HOW state is reached from state Q/ if the "current" word is 'HOW', and the predicate (NEXTCATS (ADJ SADV)) is true. If this is true, then the category of the word after the 'HOW' is adjective or adverb. In state Q/HOW, there are two arcs, a CAT ADJ arc and a CAT SADV arc, both with trivially true conditions T. Hence, the only way to reach state Q/HOW is after the word 'HOW' has been consumed and the "current" word is an adjective or an adverb. At state Q/HOW, the two arcs have as their only conditions that the word is an adjective or an adverb. Therefore, the parse is unable to block at state Q/HOW.

The condition/action pairs may be thought of as serving a dual purpose. They enable the system user to quickly examine what is expected at a state, what has been found up to that point, and, therefore, determine how the state "fits in" to the overall ATN - what its "meaning" is. With the implementation of the error function, the condition/action pairs also provide a powerful error facility that allows the user to experiment with the system more freely. The semantic component can be built gradually, with the user adding rules while using the error messages for sentences that he/she wants to be accepted as a guide.

Chapter Six

Users Manual

The semantic component of the RUS System is written in INTERLISP [Teitelman, 1975]; the ATN compiler [Burton, 1976] compiles the syntactic component into INTERLISP. These two references should be consulted for the details concerning the operation and use of the two systems. The ATN compiler was not available to us; the ATN was already in its compiled form. Therefore, the operation and use of the grammar compiler will not be discussed.

The RUS System is invoked by entering "rusparser" and pressing the return key. (The return is pressed after every command; we will assume this in the remaining discussion.) The system prompt is an integer (which allows reference to a previous command by using its number) followed by a "_" (underline). A phrase may now be parsed (using the present dictionary, which must at least contain "default" semantic rules for <NP> and <VP>, if nothing else,) by entering "P(<phrase>)". For example, to parse the title of this chapter, "P(USERS MANUAL)" would be entered. Notice that the input form need not be the "usual" LISP prefix form that consists of a list whose first element is the function. Here, the function can be named outside of the parentheses and the argument(s) to the function listed inside them. Also, the system automatically performs a carriage return/line feed once the parentheses "match up". (The symbol "]" can be used to indicate the appropriate number of right parentheses needed to end the current list.)

The syntactic component of the RUS System is "fixed" (since we only have it in the compiled form); changes are made by altering the semantic CASESTRUC case frames (or any other part of the dictionary entries). A typical instance that makes use of many of the system's features is the entry of a new sentence to those that have already been parsed. A good first step is to try to parse it using the "P" function described above. This will invoke the lexical prepass (described in Chapter Four) which will "complain" about any word in the input whose root form could not be located in the dictionary. If a word cannot be found, the prepass prompts for a correct spelling of the word or its definition. The user can add a word to the dictionary (without trying to parse a phrase containing it) by calling the function MAKEKNOWN. Entering "(MAKEKNOWN <word>)", where <word> is the word in question, will cause its current definition to be printed, or a request for its definition if one does not already exist. Two examples of basic definitions are shown below.

```
(N ((SENTENCE (NUMBER SG))))
   definition of "sentence", a singular noun

(V ((PARSE (PRESPART))))
   definition of "parsing", the present participle
           of "parse"
```

Once the root forms of all of the words in the input phrase are defined, the sentence can be parsed (using the "P" function). The phrase structure that is produced is based on the present dictionary. If this is the dictionary currently used, it contains only a few "default" semantic assignments (see Chapter Three). The CASESTRUC of any word can be changed by calling the special case editor "editc". Entering "editc (<word>)" initiates the editing of the CASESTRUC for the word <word>. If the word has no present case frame, "CASESTRUC is NIL - Use : command to provide initial CASESTRUC" is printed. The user may then define a case frame for this word by entering "(:(<case frame>))", where <case frame> is a case frame of the form described in Chapter Three. If a case frame already exists for the word, the editor is entered and any of the INTERLISP editing commands may be used to alter the form of the CASESTRUC. Once the desired case frame is entered, the input phrase can again be parsed, and the resulting changes in the phrase structure output examined.

Often, it is very useful to be able to see how the parse is proceeding. The parse trace "switch" is changed by entering "(PMS)" before parsing another phrase. If the trace option was off, it is now on, and vice versa. The parse trace causes the states, arcs, and new register values to be printed as a parse progresses. Also, a "NODE=..." line is output. This shows the words remaining in the input string as each state is entered (if a list) or the value popped from the processing of a "sub-network" (if a list within a list). The user may then examine the progression of the parse and pinpoint a problem, where the ATN is unable to interpret something the user wanted it to, for example.

In addition to this "parse trace" option, there is also an INTERLISP trace function that allows any function to be "traced". Entering "trace (<fn 1>...<fn n>)" will cause the functions <fn 1>...<fn n> to be "traced". This means that every time one of the functions is called, the value of its arguments is output, it itself is evaluated, and then the resulting value is printed. In this way the user can know what functions are called, what their arguments are, and what result the function produces. Often, if a group of functions that the user is interested in are invoked frequently during a parse (such as ASSIGN), the user might "put a trace" on the functions, set the "parse tree" switch as described above, and then parse a sample phrase. Now the user can see the progression of the parse from state to state in the ATN as well as examine the effect of the functions in question on the parse.

Another INTERLISP feature that is very useful for debugging is the break function. Typing "break (<fn 1>...<fn n>)" will cause the functions listed to be "broken" whenever they are called. This means that as one is called, but before it is evaluated, a message "(<fn x> BROKEN)" is printed, a colon (the break prompt) is printed, and execution stopped. The user may now enter any valid INTERLISP command (just as if he/she were at the "normal" level of processing) or one of a numer of break commands. Some of the more useful commands are listed below.

STATE	prints the current state in the ATN
?=	prints the values of the arguments of the broken function
EVAL	evaluates the broken function
VALUE	prints the value of the break (which is the value of the function if it was just EVALed)
GO	evaluates the broken function (if it was not already EVALed), prints the value of the break, and then ends the break (i.e., continues processing)
OK	same as GO except the value is not printed.

The effects of both the "trace" and "break" debugging commands are removed by the "unbreak" command. Entering "unbreak (<fn 1>...<fn n>)" will remove the listed functions from the trace and break lists; "unbreak ()" will remove all breaks and traces. Both the trace and break functions are much more flexible than shown here. The full list of options available when using them are explained in the INTERLISP Manual.

Once an input phrase has been parsed, the result is placed in the variable "**". Although a phrase structure is always printed as the result of the parse, the structure upon which it is based can also be examined. Typing "editv (**)" will allow the user to edit the full output of the parse. Any editing command can then be entered. Since the reason for editing "**" is usually to examine it, not change it, the "pp" editing command to "prettyprint" the value of "**" is usually the only command entered (other than "ok" to exit the editor, that is). The variable DON'TPRINTSLOTS holds a list of slots in "**" that are not to be printed in the normal parser output. Changing this variable's value is the way to alter the form of the parser output that is automatically printed as the result of the parse; the value of "**" is unaffected.

Occassionally, it is useful to be able to save all or a portion of a terminal session on the system. In this way, many people may be able to examine a sample session more easily, and the actual printout from the session can be lost without grave consequences. A "transcript" of a portion of a terminal session may be saved by entering "dribble (<filename>)". From the point where that command was entered until "dribble ()" is entered, all lines are appended to the end of the file <filename>. If the file did not exist, it is created, filled with the "dribbled" lines, and saved.

The preceding discussion is meant to serve only as an introduction to allow a user somewhat familiar with LISP editors, but unfamiliar with INTERLISP and the RUS System, to experiment with the system. Many more options exist for the commands described above; and the INTERLISP system has many additional commands that the user might find useful in experimenting with the RUS System.

Chapter Seven

Conclusion

The optimal methodology for a natural language processing system is one in which the one of the two components (syntax or semantics) that is "in control" at any given time is the one that is best able to determine how the current portion of the input should be handled. Semantic grammars achieved this integration of the two types of "knowledge", but at the cost of a decrease in flexibility and a near total lack of transportability to new application domains. The RUS System has frequent interaction between the two components (as soon as the "head" of a syntactic unit is found), but has completely separate representations for them; moving to a new application domain requires altering only the semantic component.

The Augmented Transition Network that is used was developed while being applied to three different domains [Bobrow, 1978]. The resulting parser handles a very large subset of English; many of the constructs that were missed in one application were found while examining another. Also, the Well-Formed Substring Table facility, GROUP arcs, "lookahead" functions, and altered depth-first control structure have increased the determinism of the "standard" ATN paradigm. These attempts to move away from "hypothesis-driven" toward "data-driven" processing, along with the ATN being compiled have greatly improved the efficiency of the ATN. Actual implementations of RUS-based systems have parsed from fifty to sixty percent of their inputs with no back-up, and from fifteen to twenty-five percent with "semantic back-up" (due to interpreting a construct correctly, but at the wrong level - as people would do in a left-to-right examination of the input) [Mark and Barton, 1980; Bobrow, 1978]. Since the syntactic component of the system is "fixed", it is crucial that it is able to process a large subset of English efficiently. (Fairly complex sentences took three-fourths of a second of CPU time to parse.)

A "usable" natural language processing system must have an error message facility that is invoked when a parse fails. The "state meanings" concept has been implemented in an earlier system [Black, 1979; Weischedel and Black, 1979]. It was found to be applicable to a general class of grammars and to require little modification of the grammar itself. Because of the difficulties that may arise in the writing of the semantic rules for a new application domain, the error message facility would be an invaluable aid in determining the cause of parsing failures. Indeed, the constructing of a set of semantic rules by a non-experienced designer could be a very difficult process without such a facility. Although the functions to implement this facility for the RUS System have not yet been written, the moderate effort we expended in writing the "meanings of the states", along with the results of the previous work, indicate that writing the necessary implementation functions can be completed.

Although a RUS-oriented semantic component (the dictionary) was not available, those who have developed (or are planning to develop) a lexicon for a new domain have estimated the effort involved to be from one to four person-months [Mark and Barton, 1980; Bobrow, 1978]. This represents a significant improvement in the degree of transportability of natural language processing systems.

The closer the syntactic and semantic components of a system work together, the more efficiently the system will process English input. Although the RUS System utilizes the method of frequent interactions based on the finding of syntactic "units", the best interaction methodology intuitively seems to be to allow the two components to process an input phrase totally "in parallel" (with the component with the most "knowledge" at any given point in control). However, the communication/control problems in multiprocessing are difficult ones; it is quite possible that the overhead needed to allow such processing would strain the efficiency of the system more than creating the "optimal parsing environment" would improve it.

The RUS System meets many of the goals of a good natural language processing system. The separation of the syntactic and semantic components allows the capturing of syntactic regularities and the modification of one type of "knowledge" separately from the other. The frequent semantic checks permit substantial guidance of the parse by domain-dependent "knowledge". The addition of many features to the standard ATN formalism has greatly increased its power and efficiency. The case structure representation of domain "knowledge" is a very flexible representation; the user may "define" the ways in which a word may be used as broadly or as narrowly as desired. In comparison to previous natural language processing systems, the RUS System is able to be applied to a wider range of application domains, has a greater flexibility in how domain-dependent "knowledge" can be represented, uses the domain "knowledge" to a significant degree in guiding the parse, and is efficient enough to respond to input phrases within a one second time period.

Bibliography

Black, John E. "Generating Error Messages for Naive Users of Software Having Natural Language Input." Unpublished Masters thesis, University of Delaware, 1979.

Bobrow, Robert J. "The RUS System." _Research in Natural Language Understanding_. BBN Report No. 3878. Cambridge, Massachusetts: Bolt Beranek and Newman Inc., 1978.

Bruce, Bertram C. "Case Systems for Natural Language." _Artificial Intelligence_, December, 1975, 327-360.

Burton, Richard R. _Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems_. BBN Report No. 3453. Cambridge, Massachusetts: Bolt Beranek and Newman Inc., 1976.

Mark, William S. and Barton, G. Edward, Jr. _The RUSGRAMMAR Parsing System_. Research Publication GMR-3243. Warren, Michigan: General Motors Research Laboratories, 1980.

Teitelman, Warren _INTERLISP Reference Manual_. Palo Alto, California: Xerox Palo Alto Research Center, 1975.

Weischedel, Ralph M. and Black, John E. _Responding to Potentially Unparsable Sentences_. Technical Report No. 79/3, Department of Computer and Information Sciences, University of Delaware, Newark, Delaware, 1979. (To appear in _American Journal of Computational Linguistics_, 1980).

Winograd, Terry. "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language." Doctoral dissertation, Massachusetts Institute of Technology, 1971.

Woods, William A. "An Experimental Parsing System for Transition Network Grammars." _Natural Language Processing_. Edited by R. Rustin. New York: Algorithmics Press, 1973.

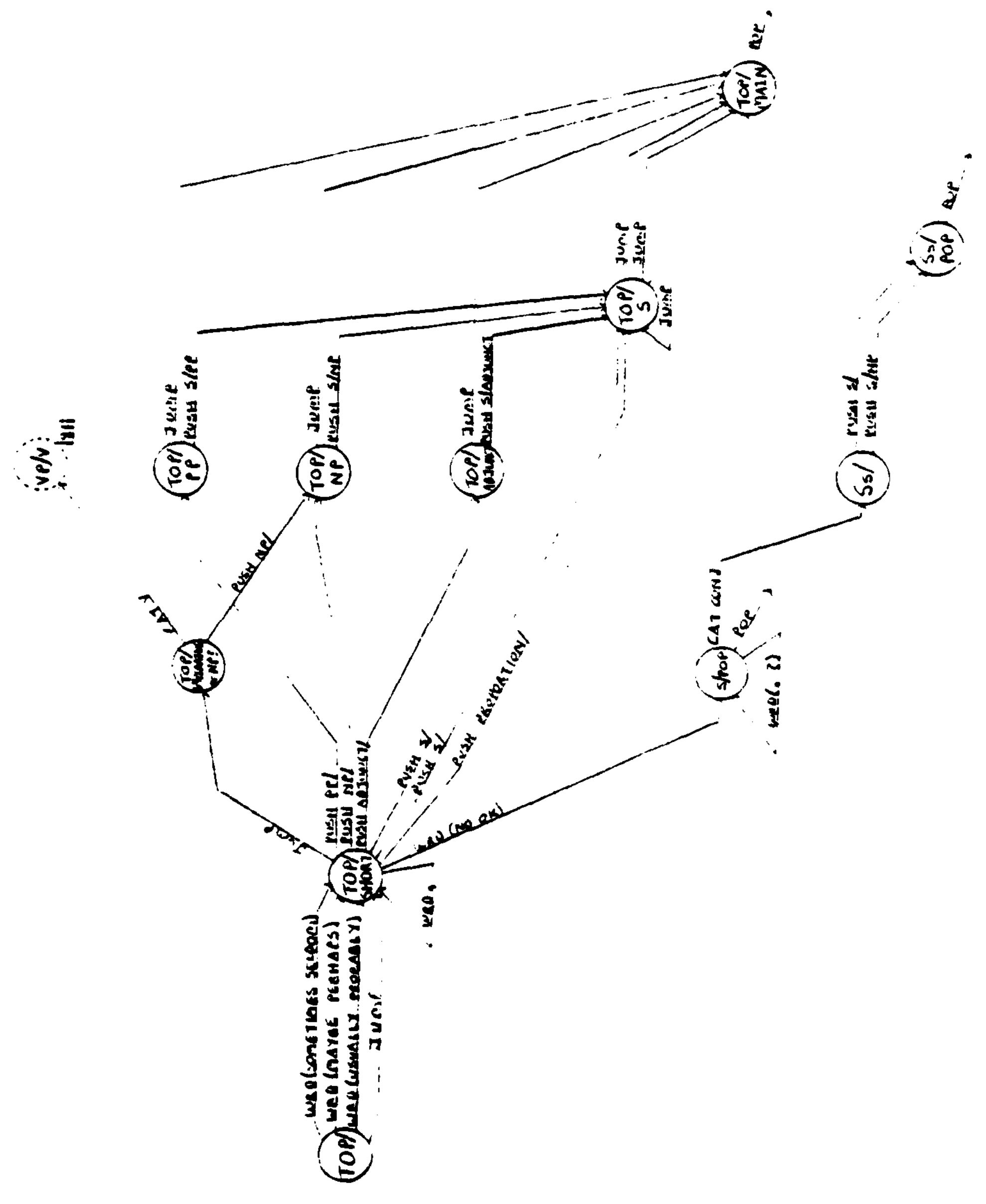
Woods, William A. "Semantics and Quantification in Natural Language Question Answering." Advances in Computers. Vol. 17. Edited by M. C. Yovits. New York: Academic Press, 1978.

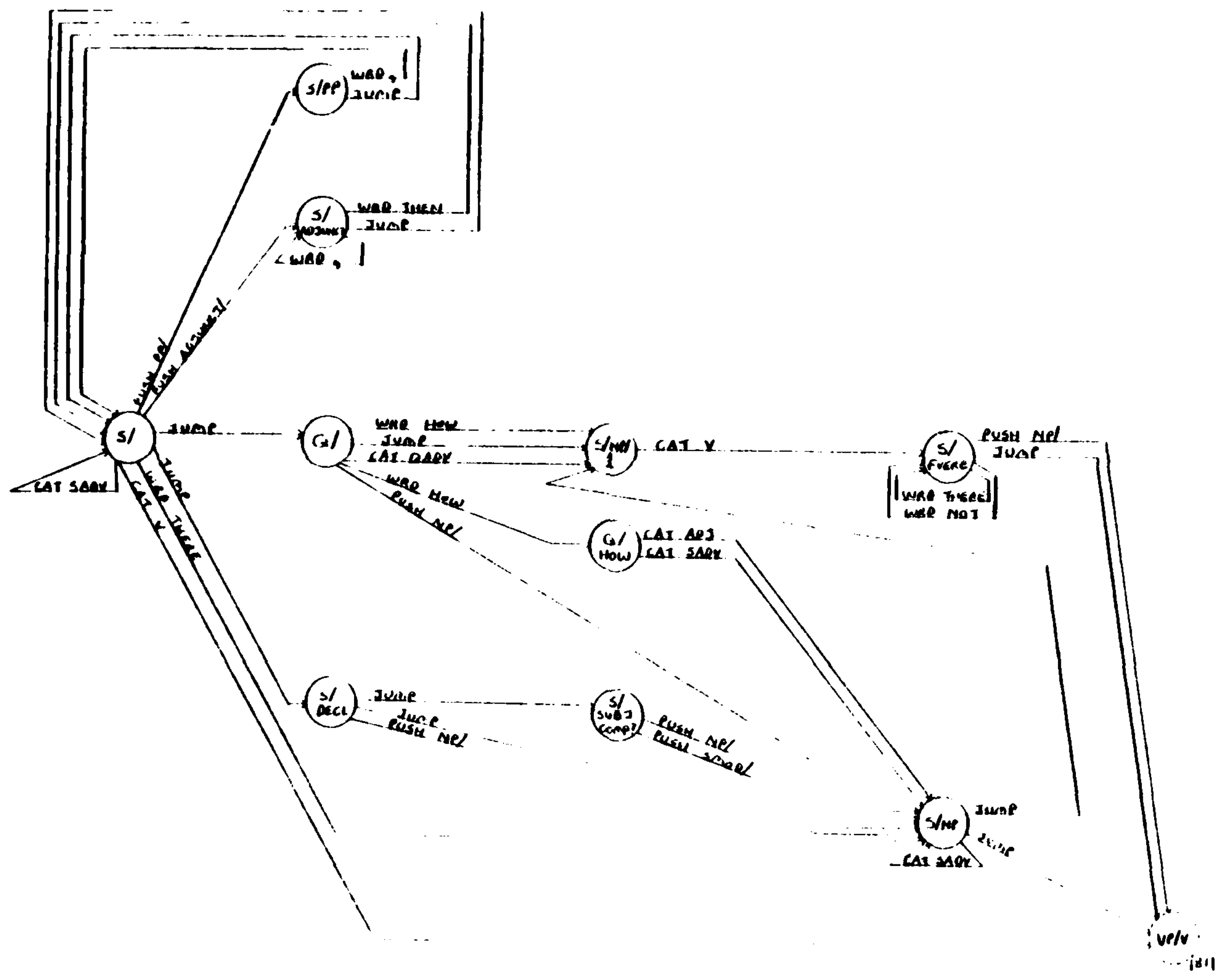
Appendix A

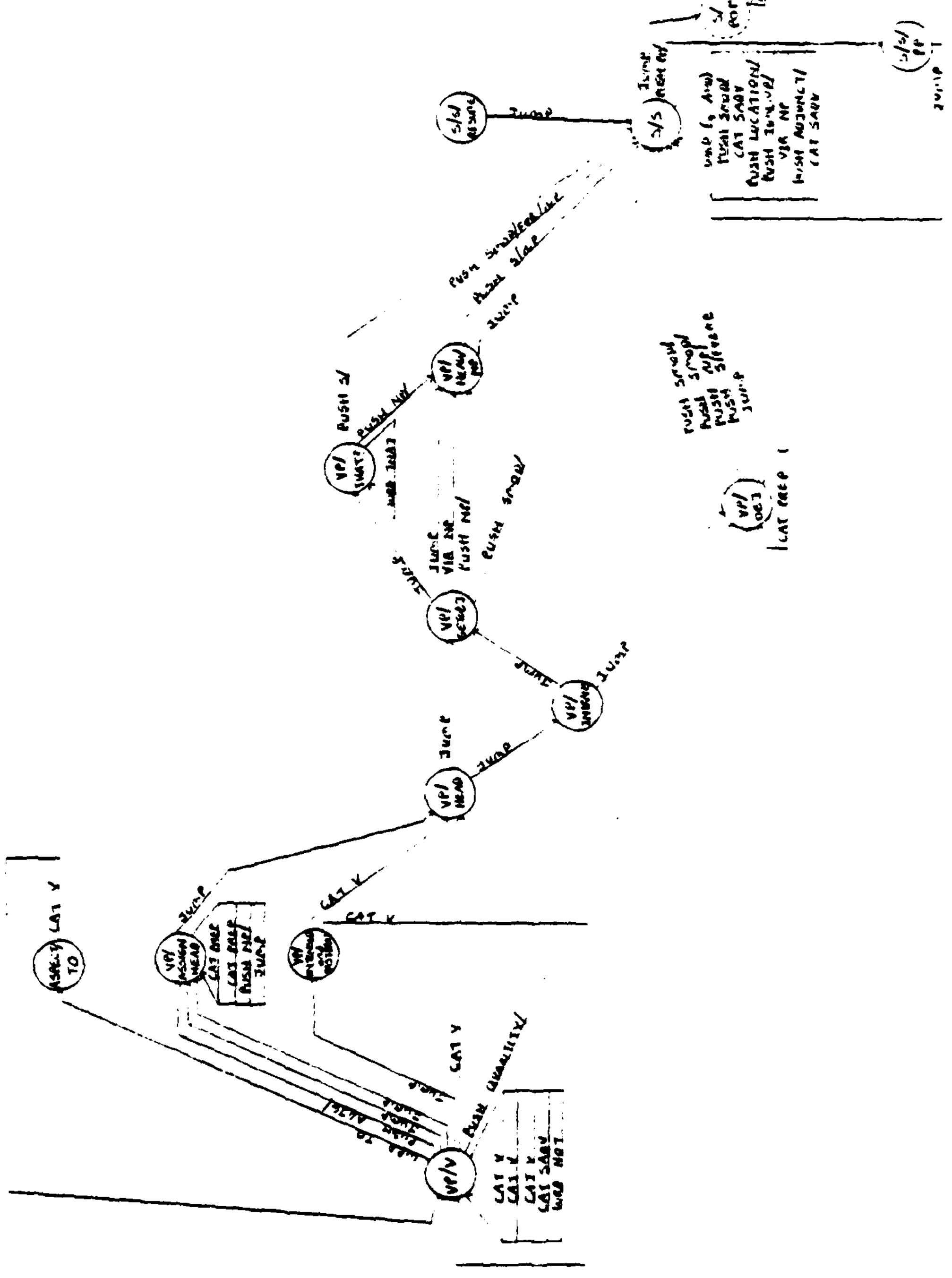
RUS ATN Graph

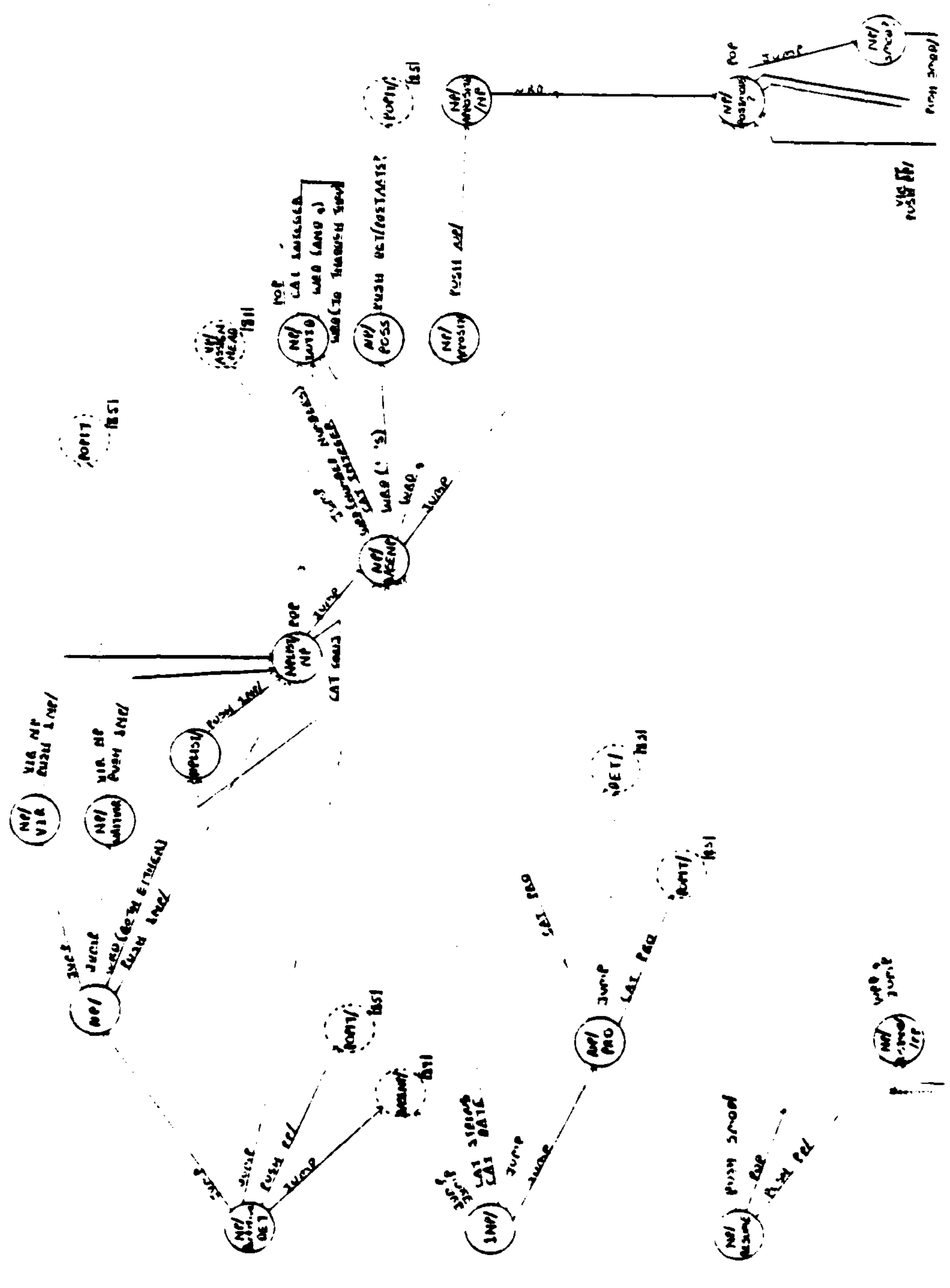
The diagrams on the following pages represent the Augmented Transition Network of the RUS System. The full list of conditions (and actions) are impossible to include in such a diagram. The arc labels do include the "type" of the arc (WRD, CAT, JUMP, etc.) and its arguments. For example, a CAT arc label includes the category being looked for. None of the additional conditions or any of the actions are listed.

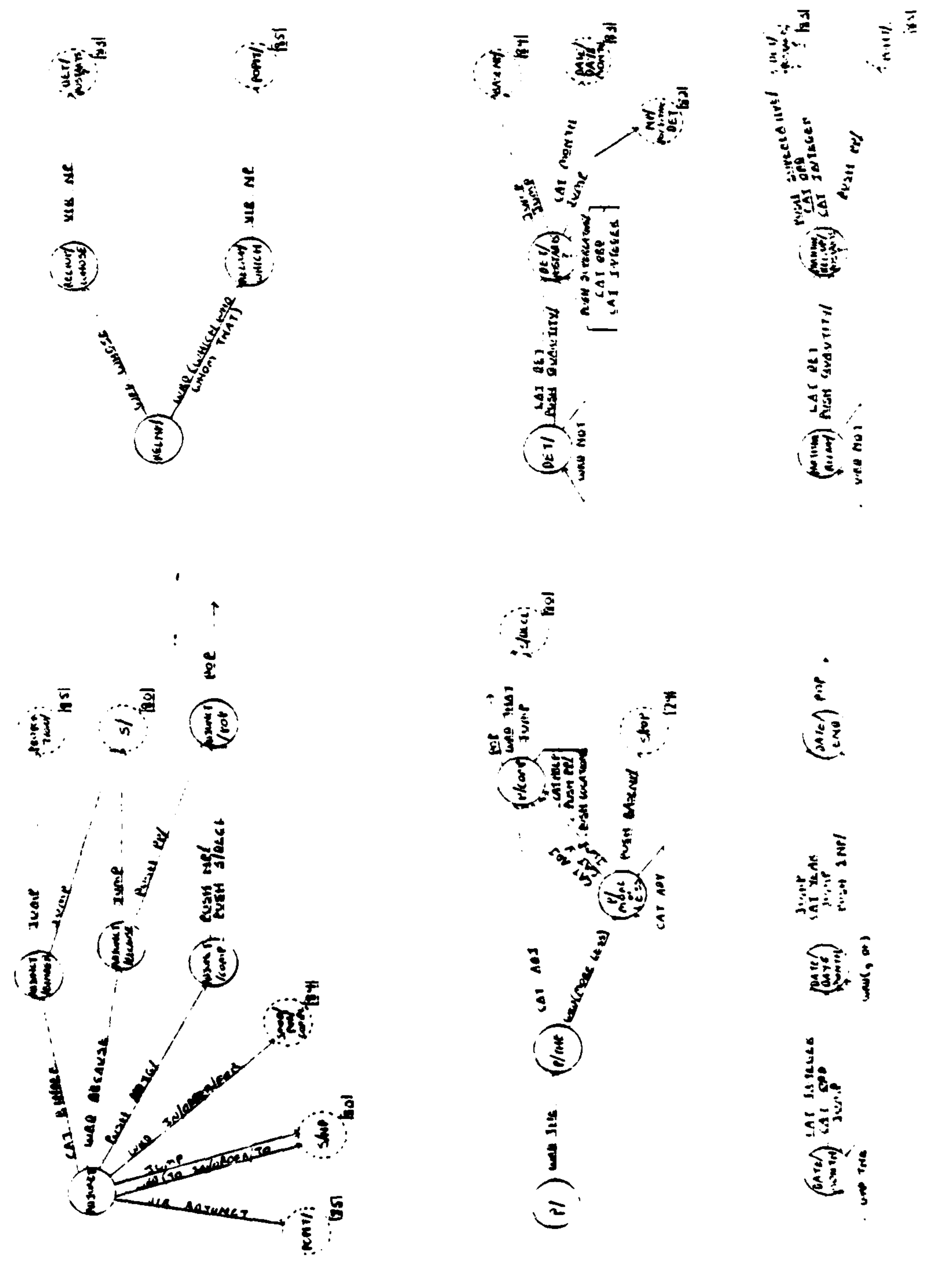
If a state is represented by dotted lines, that indicates that the state appears on another page of the graph as well. The number in the square next to the dotted line state indicates the page number on which the state appears in "non-dotted" form.

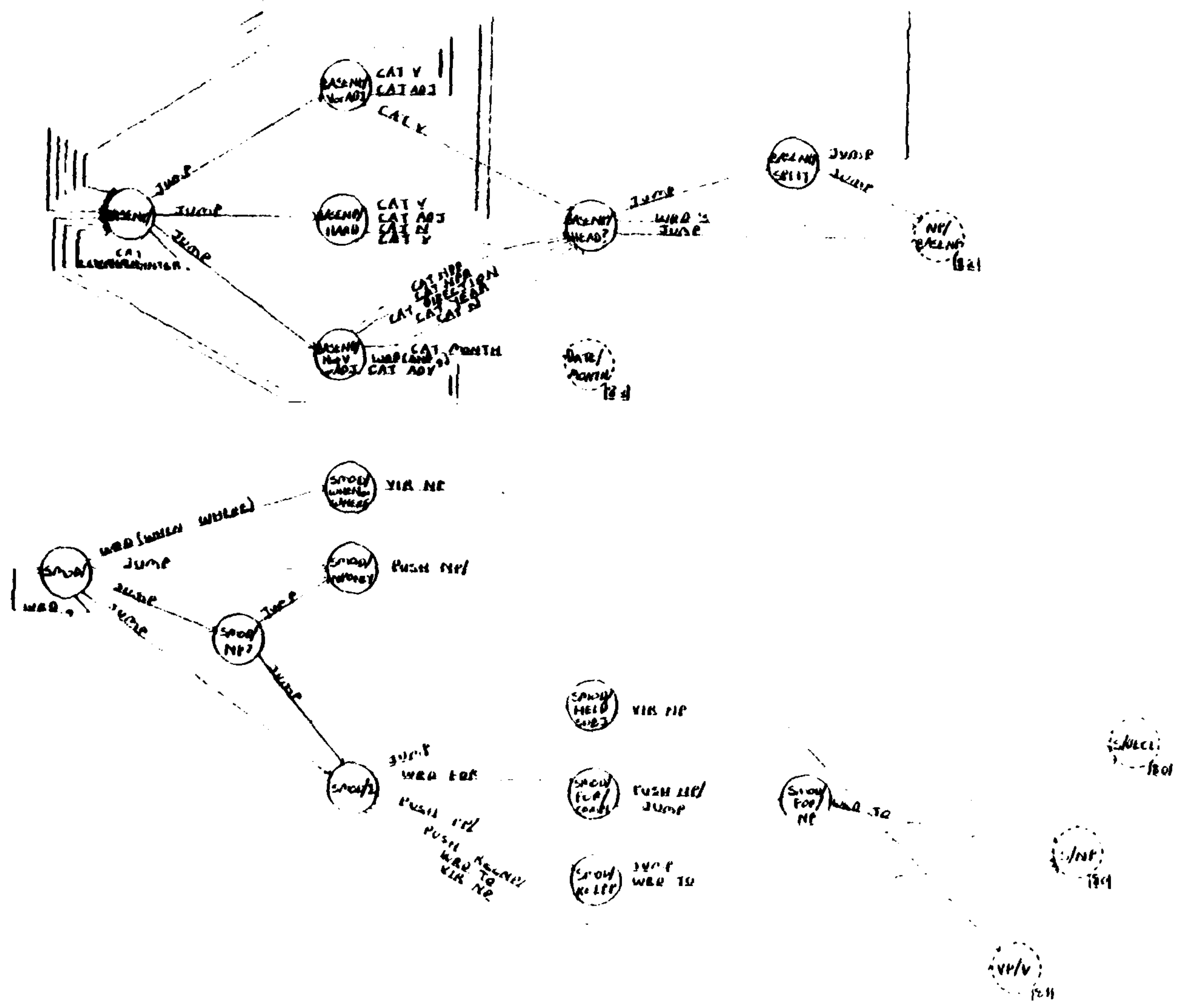

TOP/
WRD (SOMETIMES SELDOM)
WRD (MAYBE PERHAPS)
WRD (USUALLY PROBABLY)
JUMP
TOP/ SHORT
WRD,
JUMP
PUSH PP/
PUSH NP/
PUSH ADJUNCT/
PUSH S/
PUSH S/
PUSH EXPLORATION/
WRD (NO OK)
TOP/ NP!
CAT V
PUSH NP/
VP/V
TOP/ PP
JUMP
PUSH S/PP
TOP/ NP
JUMP
PUSH S/NP
TOP/ ADJUNCT
JUMP
PUSH S/ADJUNCT
TOP/ S
JUMP
JUMP
JUMP
TOP/ MAIN
POP,
S/POP
CAT CONJ
POP
WRD(, ?)
Ss/
PUSH S/
PUSH S/NP
Ss/ POP
POP

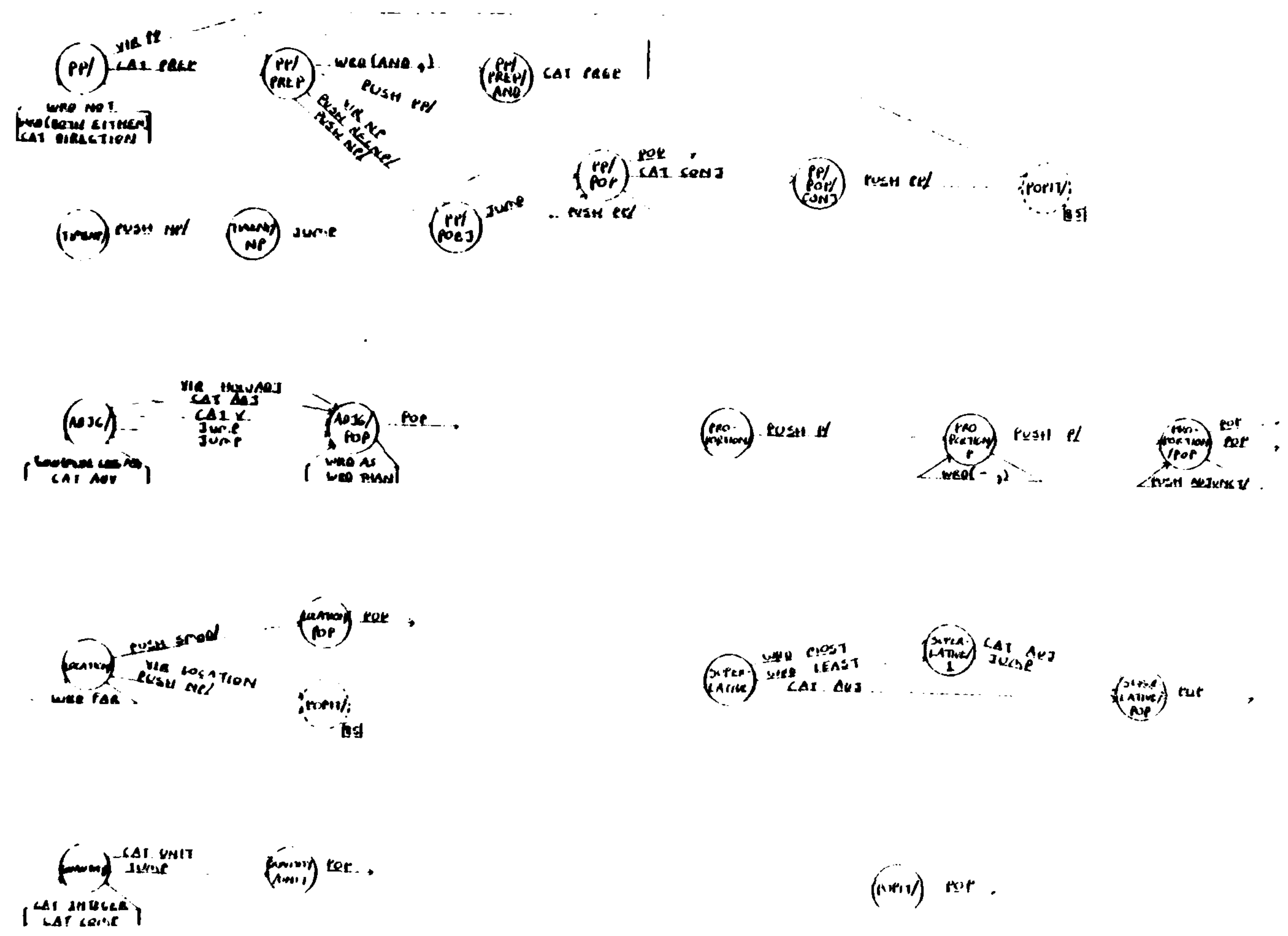

S/
S/PP
WRD ,
JUMP
S/ ADJUNCT
WRD THEN
JUMP
< WRD ,
PUSH PP/
PUSH ADJUNCT/
JUMP
CAT SARY
JUMP
WRD THERE
CAT V
Q/
WRD HOW
JUMP
CAT DARY
WRD HOW
PUSH NP/
S/NP
CAT V
Q/ HOW
CAT ADJ
CAT SARY
S/ THERE
PUSH NP/
JUMP
WRD THERE
WRD NOT
S/ DECL
JUMP
JUMP
PUSH NP/
S/ SUBJ COMP
PUSH NP/
PUSH SCOMP/
S/NP
JUMP
JUMP
CAT SARY
VP/V


VP/V
CAT V
JUMP
PUSH NP/
PUSH QUANTITY/
VP/HEAD
S/S
CAT PREP






CAT V
CAT ADJ
CAT V
CAT ADJ
CAT N
CAT V
CAT MONTH
CAT ADV
JUMP
WRD 'S
JUMP
NP/
VIR NP
PUSH NP/
JUMP
WRD FOR
PUSH NP/
VIR NP
PUSH NP/
JUMP
WRD TO
/NP
VP/V


PP/
VIR PP
CAT PREP
WRD NOT
WRD (BOTH EITHER)
CAT DIRECTION
PP/PREP
WRD (AND ,)
PUSH PP/
VIR NP
PUSH NP/
PP/PREP/AND
CAT PREP
PP/POP
POP
CAT CONJ
PUSH PP/
PP/POP/CONJ
PUSH PP/
TIME/
PUSH NP/
TIME/NP
JUMP
PP/POBJ
JUMP
ADJC/
VIR
CAT ADJ
CAT V
JUMP
JUMP
ADJC/POP
POP
WRD AS
WRD THAN
CAT ADV
PRO-PORTION
PUSH
PUSH
POP
POP
LOCATION
PUSH
VIR LOCATION
PUSH NP/
WRD FAR
POP
SUPERLATIVE
WRD MOST
WRD LEAST
CAT ADJ
CAT ADJ
JUMP
POP
CAT UNIT
JUMP
CAT INTEGER
CAT CONJ
POP
POP

Appendix B

State Meanings:
S, VP, and NP Groups

```
(S/
  ((AND (CAT V)
        (CHECKF V UNTENSED)
        (NOT (** : IMPERATIVE)))
   (; An imperative is expected.  The verb must be untensed
           and have a CASESTRUC rule for imperative.  In this
           case, it is untensed but has no rule for imperative.)
   (PCHAR (QUOTE " THE VERB ' "))
   (PRINT-ANY-STRING LEX)
   (PCHAR (QUOTE " ' , ALTHOUGH UNTENSED, CANNOT BE UNDERSTOOD "))
   (PCHAR (QUOTE " AS AN IMPERATIVE. "))
   (TERPRI))
  ((AND (CAT V)
        (NOT (CHECKF V UNTENSED))
        (NOT (** : IMPERATIVE)))
   (; As in the case above, except that the verb is not untensed.)
   (PCHAR (QUOTE " THE VERB ' "))
   (PRINT-ANY-STRING LEX)
   (PCHAR (QUOTE " ' CANNOT START A SENTENCE.  A VERB HERE "))
   (PCHAR (QUOTE " INDICATES AN IMPERATIVE AND MUST BE UNTENSED. "))
   (TERPRI))
  ((CAT V)
   (; As above, except the verb has an imperative rule, but is not
           untensed.)
   (PCHAR (QUOTE " TO BE AN IMPERATIVE, THE VERB MUST BE UNTENSED. "))
   (TERPRI))
  (T
   (; Blocked at a word that is not a verb.  List of what is
           expected at this state is printed out.)
   (PCHAR (QUOTE " THE BEGINNING OF A SENTENCE (OR AN EMBEDDED "))
   (PCHAR (QUOTE " SENTENCE) MUST BE ONE OF THE FOLLOWING:  "))
   (PCHAR (QUOTE "          1.  AN ADVERB                   "))
   (PCHAR (QUOTE "          2.  A PREPOSITION               "))
   (PCHAR (QUOTE "          3.  AN ADJUNCTIVE PHRASE        "))
   (PCHAR (QUOTE "          4.  AN UNTENSED VERB            "))
   (PCHAR (QUOTE "          5.  A QUESTION VERB, ADVERB, OR "))
   (PCHAR (QUOTE "                DETERMINER                "))
   (PCHAR (QUOTE "          6.  THE WORD 'THERE'            "))
   (PCHAR (QUOTE "          7.  A NOUN PHRASE               "))
   (TERPRI)))
```

```
(Q/
   ((WRD HOW)
    (; In a question, the 'HOW' must be followed by an adjective,
            adverb, or verb.)
    (PCHAR (QUOTE " THE WORD 'HOW' MUST BE FOLLOWED BY AN      "))
    (PCHAR (QUOTE " ADJECTIVE, AN ADVERB, OR A VERB, AS IN:    "))
    (PCHAR (QUOTE "         HOW BIG IS...                      "))
    (PCHAR (QUOTE "         HOW QUICKLY CAN...                 "))
    (PCHAR (QUOTE "         HOW DID THEY...                    "))
    (TERPRI)))


(S/NP/1
   ((AND (CAT V)
         (EQ (GETR COMPLTYPE)
             (QUOTE FOR)))
    (; A 'FOR' Complement is expected.  The verb must be a present
            participle.)
    (PCHAR (QUOTE " IN A 'FOR COMPLEMENT', THE VERB MUST BE A "))
    (PCHAR (QUOTE " PRESENT PARTICIPLE, AS IN:                "))
    (PCHAR (QUOTE "         ...FOR 'PRINTING' MESSAGES...     "))
    (TERPRI))
   ((AND (CAT V)
         (EQ (GETR COMPLTYPE)
             (QUOTE TO)))
    (; A 'TO' Complement is expected.  The verb must be untensed.)
    (PCHAR (QUOTE " IN A 'TO COMPLEMENT', THE VERB MUST BE AN "))
    (PCHAR (QUOTE " INFINITIVE, AS IN:                        "))
    (PCHAR (QUOTE "         ...TO 'BE' AN ERROR...            "))
    (TERPRI))
   ((CAT V)
    (; If we do not have a complement as in the cases above, the
            verb must not be a present participle.)
    (PCHAR (QUOTE " THE VERB HERE CAN BE ANY FORM OTHER "))
    (PCHAR (QUOTE " THAN A PRESENT PARTICIPLE. "))
    (TERPRI))
   ((EQ (CAR (GETR ?))
        (QUOTE QADV))
    (; We do not have a verb.  (The previous three conditions
            handled all verb possibilities.)  And we just had a
            question-adverb, such as 'WHEN'.)
    (PCHAR (QUOTE " IN A QUESTION, THE WORD ' "))
    (PRINT-ANY-STRING (CDR (GETR ?)))
    (PCHAR (QUOTE " ' SHOULD BE FOLLOWED BY A VERB. "))
    (TERPRI))
   (T
    (; Expecting a verb.)
    (PCHAR (QUOTE " A VERB FORM IS EXPECTED HERE. "))
    (TERPRI)))
```

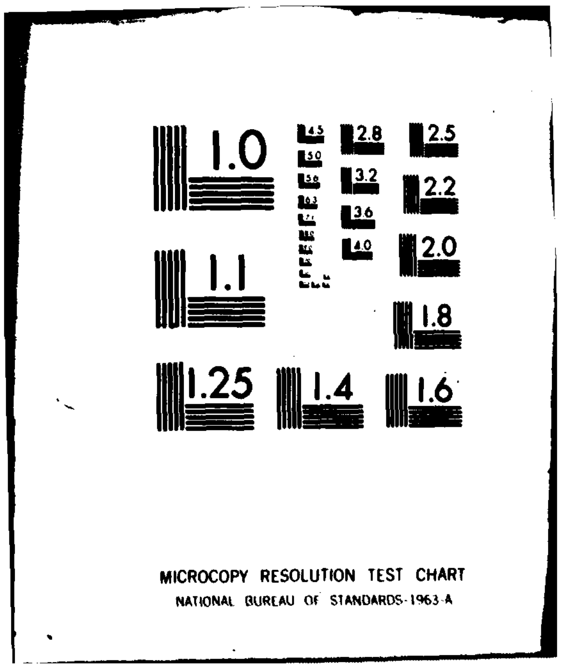
```
 (; A 'NOT' is valid here only if an adverb has been encountered.
       A 'NOT' before the adverb has already been taken care of
       in state S/FVERB.)
 (PCHAR (QUOTE " THE 'NOT' HERE DOES NOT MAKE SENSE.  A "))
 (PCHAR (QUOTE " 'NOT' CAN BE HERE ONLY IF AN ADVERB "))
 (PCHAR (QUOTE " HAS BEEN FOUND. "))
 (TERPRI))
((AND (CAT V)
      (CHECKF V PRESPART)
      (NOT (ASPECTVERB HEAD)))
 (; A present participle here is alright as long as the head verb
       is an aspect verb.  (e.g., 'IS GOING', 'BEGIN RUNNING'))
 (PCHAR (QUOTE " THE PRESENT PARTICIPLE ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD BECAUSE IT MUST BE "))
 (PCHAR (QUOTE " PART OF AN ASPECT VERB PHRASE, SUCH AS:   "))
 (PCHAR (QUOTE "         HE BEGAN 'RUNNING'...             "))
 (PCHAR (QUOTE "         I WILL START 'SINGING'...         "))
 (TERPRI)
 (PCHAR (QUOTE " THE VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' IS NOT A ASPECT VERB. "))
 (TERPRI))
((AND (CAT V)
      (CHECKF V UNTENSED)
      (NULL (CDR (GETR ASPECT)))
      (NULL (GETR MODAL)))
 (; An untensed verb here is alright only if we had a modal.
       e.g., I MIGHT 'GO'...)
 (PCHAR (QUOTE " FOR AN UNTENSED VERB SUCH AS ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ' TO BE HERE, WE MUST HAVE HAD A "))
 (PCHAR (QUOTE " MODAL VERB, SUCH AS:              "))
 (PCHAR (QUOTE "      I 'MIGHT' GO...              "))
 (PCHAR (QUOTE "      HE 'WILL' EXPLAIN...         "))
 (TERPRI))
((AND (CAT V)
      (OR (EQREG HEAD BE)
          (EQREG HEAD GET)
          (EQREG HEAD BECOME))
      (CHECKF V PASTPART)
      (NOT (VPASSIVE *)))
 (; If we had a form of 'BE', 'GET', or 'BECOME', the past
       participle here should be part of a passive construction.
       e.g., IT WAS 'SENT'...)
 (PCHAR (QUOTE " BECAUSE OF THE HEAD VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' AND THIS PAST PARTICIPLE ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ', A PASSIVE CONSTRUCTION IS EXPECTED. "))
 (PCHAR (QUOTE " HOWEVER, THE PAST PARTICIPLE VERB HERE "))
 (PCHAR (QUOTE " IS NOT ABLE TO BE PART OF A PASSIVE PHRASE. "))
 (TERPRI))
((AND (CAT V)
      (OR (EQREG HEAD BE)
          (EQREG HEAD GET)
```

AD-A091 043 DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
A CASE STUDY IN NATURAL LANGUAGE PROCESSING: THE RUS SYSTEM.(U)
MAY 80 A W MANSKY F49620-79-C-0131

UNCLASSIFIED AFOSR-TR-80-1035 NL

2 of 2

END
DATE
FILMED
12-80
DTIC



```
          (EQREG HEAD BECOME))
       (VPASSIVE *)
       (NOT (CHECKF V PASTPART)))
 (; As in the above case, except that the verb is marked as
        being able to be in a passive construction, but it is
        not marked as being a past participle.)
 (PCHAR (QUOTE " A PASSIVE CONSTRUCTION IS EXPECTED.  THE "))
 (PCHAR (QUOTE " HEAD VERB BEING ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' AND THE VERB '  "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " 'BEING ABLE TO BE PASSIVE INDICATES THAT THE "))
 (PCHAR (QUOTE " LATTER SHOULD BE A PAST PARTICIPLE, BUT "))
 (PCHAR (QUOTE " IT IS UNABLE TO BE INTERPRETED AS ONE. "))
 (TERPRI))
((AND (CAT V)
      (EQREG HEAD HAVE)
      (NULL (CDR (GETR ASPECT)))
      (NULL (CHECKF V PASTPART)))
 (; If there has been a form of the verb 'HAVE', a past
        participle is expected.)
 (PCHAR (QUOTE "SINCE THERE WAS A FORM OF THE VERB "))
 (PCHAR (QUOTE " 'HAVE', A PAST PARTICIPLE IS EXPECTED HERE, "))
 (PCHAR (QUOTE " AS IN:                                "))
 (PCHAR (QUOTE "     HE HAS 'GONE'...                  "))
 (PCHAR (QUOTE "     I HAVE 'SEEN'...                  "))
 (TERPRI)
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " IS NOT A PAST PARTICIPLE "))
 (TERPRI))
((AND (OR (CAT ADJ)
          (WRD (AS MORE LESS)))
      (NOT (COPULA HEAD)))
 (; An adjective or one of 'AS', 'MORE', or 'LESS' indicates the
        beginning of an adjective group if the head verb was a
        copula.  e.g., IT BECAME 'MORE' THAN... THIS IS 'AS'
        LARGE A PROBLEM...)
 (PCHAR (QUOTE " THE WORD ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ' INDICATES THE BEGINNING OF AN ADJECTIVE  "))
 (PCHAR (QUOTE " GROUP, BUT THE HEAD VERB MUST BE A COPULA- "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " IS NOT ONE.  EXPECTING PHRASES OF THE FORM: "))
 (TERPRI)
 (PCHAR (QUOTE "      IT IS "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ... "))
 (TERPRI)
 (PCHAR (QUOTE " WHERE THE WORD ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ' IS THE BEGINNING OF AN ADJECTIVE PHRASE. "))
 (TERPRI))
(T
 (; Blocked at this state for other reasons.  Because there are
        so many arcs (12) and conditions (24) at this state,
        only the eight (most likely?) conditions above are
       checked.  If none of those are applicable, a list of
       what was expected is printed.)
(PCHAR (QUOTE " AT THIS POINT IN THE SENTENCE, ONE OF "))
(PCHAR (QUOTE " THE FOLLOWING WAS EXPECTED. "))
(TERPRI)
(PCHAR (QUOTE "   1. AN ADJECTIVE, 'AS', 'MORE', OR 'LESS'      "))
(PCHAR (QUOTE "           FOLLOWING A COPULA VERB,              "))
(PCHAR (QUOTE "           (IT IS 'LARGER' THAN...)              "))
(PCHAR (QUOTE "           (THEY SEEMED 'MORE' HAPPY...)         "))
(PCHAR (QUOTE "   2. 'TO' FOLLOWING AN ASPECTYPE VERB,          "))
(PCHAR (QUOTE "           (WE STARTED 'TO' LEAVE...)            "))
(PCHAR (QUOTE "   3. AN UNTENSED PAST PARTICIPLE,               "))
(PCHAR (QUOTE "           ('COME', 'RUN', ETC.)                 "))
(PCHAR (QUOTE "   4. AN UNTENSED VERB FOLLOWING A MODAL,        "))
(PCHAR (QUOTE "           (THEY SHOULD 'GO'...)                 "))
(PCHAR (QUOTE "           (I WILL 'SHOW'...)                    "))
(PCHAR (QUOTE "   5. A PRESENT PARTICIPLE FOLLOWING AN          "))
(PCHAR (QUOTE "           ASPECTYPE VERB,                       "))
(PCHAR (QUOTE "           (THEY BEGAN 'RUNNING'...)             "))
(PCHAR (QUOTE "   6. A PAST PARTICIPLE IN A PASSIVE             "))
(PCHAR (QUOTE "           CONSTRUCTION AFTER 'BE', 'GET',
(PCHAR (QUOTE "           OR 'BECOME',                          "))
(PCHAR (QUOTE "           (IT WAS 'GIVEN'...)                   "))
(PCHAR (QUOTE "           (THEY ARE 'AMUSED' BY...)             "))
(PCHAR (QUOTE "   7. A PAST PARTICIPLE FOLLOWING 'HAVE',        "))
(PCHAR (QUOTE "           (HE HAS 'GIVEN' EVERYTHING...)        "))
(PCHAR (QUOTE "           (I HAVE 'CANCELLED' THE...)           "))
(PCHAR (QUOTE "   8. AN INTEGER OR COMPARATIVE FOLLOWING 'BE', "))
(PCHAR (QUOTE "           (THERE ARE 'FIVE'...)                 "))
(PCHAR (QUOTE "   9. AN ADVERB.                                 "))
(PCHAR (QUOTE "           (HE RAN 'QUICKLY'...)                 "))
(TERPRI)
(PRINT-ANY-STRING LEX)
(PCHAR (QUOTE " IS NOT ABLE TO BE INTERPRETED AS ANY OF THESE. "))
(TERPRI)))
```

```
(ASPECT/TO
   ((AND (CAT V)
         (NOT (CHECKF V UNTENSED)))
    (; ASPECT VERB followed by 'TO' should then be followed by an
           untensed verb.  This one is tensed.)
    (PCHAR (QUOTE " AN 'ASPECT' VERB, SUCH AS ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ', FOLLOWED BY 'TO', EXPECTS AN UNTENSED VERB "))
    (PCHAR (QUOTE " TO BE NEXT, AS IN:                            "))
    (TERPRI)
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " TO LEAVE... "))
    (TERPRI)
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " IS NOT UNTENSED. "))
    (TERPRI))
   ((NOT (CAT V))
    (; As above, except we don't even have a verb.)
    (PCHAR (QUOTE " AN 'ASPECT' VERB FOLLOWED BY 'TO' EXPECTS AN  "))
    (PCHAR (QUOTE " UNTENSED VERB NEXT, AS IN:                    "))
    (TERPRI)
    (PCHAR (QUOTE "          IT BEGAN TO 'GROW'.                  "))
    (TERPRI)
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " IS NOT A VERB. "))
    (TERPRI)))


(VP/ASSIGNHEAD
   ((CAT PREP)
    (; A preposition is permitted here if the verb can take one as
           part of the verb ('START UP' THE CAR.) or as an object
           of the verb.  (WE LOOKED 'UP'.))
    (PCHAR (QUOTE " A PREPOSITION SUCH AS ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' CAN BE UNDERSTOOD HERE ONLY IF THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' CAN TAKE A PREPOSITION AS A VERB PART, AS "))
    (PCHAR (QUOTE " IN 'START UP', OR AS AN OBJECT, AS IN "))
    (PCHAR (QUOTE " 'LOOK DOWN'.  NEITHER ONE IS THE CASE. "))
    (TERPRI))
   ((NOT (OR (AND (EQREG STYPE POSS-ING)
                  (ASSIGNQ HEAD
                       (OR (GetPATHR HEAD HEAD: PARTICIPLE/OF)
                           (GetPATHR HEAD HEAD))
                        HEAD VERBCASES))
             (ASSIGNQ HEAD HEAD HEAD VERBCASES)
             (ForceParseFLG)))
    (; Hoping to go get objects after finishing verb, but verb does
           not check out semantically.)
    (PCHAR (QUOTE " THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE MAIN VERB "))
    (PCHAR (QUOTE " IN THE PHRASE. "))
    (TERPRI))
   ((NOT (OR (EQREG STYPE IMPERATIVE)
          (EQREG STYPE POSS-ING)
          (AND (GETR SUBJCOMP)
               (ASSIGNQ SUBJCOMP HEAD SUBJCOMP VERBCASES))
          (AND (ASSIGNQ FIRSTNP HEAD SUBJECT VERBCASES)
               (SETR SUBJECT FIRSTNP))
          (ForceParseFLG)))
 (; The noun phrase in register FIRSTNP does not check semantically
        with the verb in HEAD.  Also, if there is a subject
        complement in SUBJCOMP, it also does not check out
        semantically.
 (COND ((GETR SUBJCOMP)
        (PCHAR (QUOTE " THE PHRASE ' "))
        (PRINT-REG (GETR SUBJCOMP))
        (PCHAR (QUOTE " ' IS UNABLE TO BE UNDERSTOOD AS A "))
        (PCHAR (QUOTE " SUBJECT COMPLEMENT WHEN USED WITH THE VERB ' "))
        (PRINT-REG (GETR HEAD))
        (PCHAR (QUOTE " ' . "))
        (TERPRI)))
 (PCHAR (QUOTE " THE NOUN PHRASE ' "))
 (PRINT-REG (GETR FIRSTNP))
 (PCHAR (QUOTE " ' CANNOT BE INTERPRETED AS THE SUBJECT OF A "))
 (PCHAR (QUOTE " SENTENCE WHOSE VERB IS ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
(T
 (; The above conditions handle ABORTs of paths into this state;
        if the SUSPENDs at this state are taken, the verb in HEAD
        does not make sense semantically with FIRSTNP or SUBJCOMP,
        but the ForceParse was True, so a syntactic assignment was
        forced.)
 (PCHAR (QUOTE " WITH THE VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ', THE FOLLOWING CANNOT BE UNDERSTOOD: "))
 (TERPRI)
 (COND ((GETR SUBJCOMP)
        (PRINT-REG (GETR SUBJCOMP))
        (PCHAR (QUOTE " AS A SUBJECT COMPLEMENT. ")))
       (T (PRINT-REG (GETR FIRSTNP))
          (PCHAR (QUOTE " AS THE SUBJECT. "))))
 (TERPRI)))
```

```
(VP/UNTENSEDandPASTPART
   ((AND (OR (EQREG HEAD BE)
             (EQREG HEAD GET)
             (EQREG HEAD BECOME))
         (NOT (VPASSIVE *)))
    (; If we have an untensed past participle with a HEAD of 'BE',
          'GET', or 'BECOME', then we have a passive construction -
          blocked here if participle not marked as being able to be
          passive.)
    (PCHAR (QUOTE " THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' FOLLOWED BY THE PARTICIPLE ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' INDICATES A PASSIVE CONSTRUCTION, BUT THE "))
    (PCHAR (QUOTE " PARTICIPLE CANNOT BE PASSIVE. "))
    (TERPRI))
   (T
    (; If not the above case, the passive failed because HEAD was
          not one of the above three or the modal condition (the
          alternate arc) was not met.  What was expected is
          printed.)
    (PCHAR (QUOTE " AN UNTENSED PAST PARTICIPLE, SUCH AS ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ', CAN BE UNDERSTOOD HERE ONLY IF EITHER:          "))
    (TERPRI)
    (PCHAR (QUOTE "         1. THERE IS A MODAL VERB AND NO            "))
    (PCHAR (QUOTE "                  ASPECT VERB.                      "))
    (PCHAR (QUOTE "                  (THEY 'MIGHT COME.')              "))
    (PCHAR (QUOTE "     OR  2. THE MAIN VERB IS 'BE', 'GET', OR        "))
    (PCHAR (QUOTE "                  'BECOME' AND THE PHRASE IS PASSIVE."))
    (PCHAR (QUOTE "                  (IT 'WAS SET'.)                   "))
    (TERPRI)))

(VP/GETOBJ
   ((AND (WRD (FOR TO))
         (NOT (FORTOCOMP HEAD)))
    (; We expect a FOR/TO Complement, but the verb in HEAD cannot
          take such a complement.)
    (PCHAR (QUOTE " THE WORD ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' HERE INDICATES THE BEGINNING OF A COMPLEMENT.  "))
    (PCHAR (QUOTE " SUCH AS 'FOR YOU TO FINISH.'; HOWEVER,THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " CANNOT HAVE SUCH A COMPLEMENT. "))
    (TERPRI))
   ((AND (WRD THAT)
         (NOT (THATCOMP HEAD)))
    (; If we have the word 'THAT', then the verb in HEAD should be a
          THATCOMP.  The word 'THAT' does not have to be here for a
          THATCOMP, though.  If the verb is not marked as a
          THATCOMP and we have one without the 'THAT', we can't
          tell that here.  In that case, the last condition will
          print what was expected at this state.)
    (PCHAR (QUOTE " THE 'THAT' INDICATES THE BEGINNING OF A "))
    (PCHAR (QUOTE " COMPLEMENT, BUT THE VERB ' "))
```

```
     (PRINT-REG (GETR HEAD))
     (PCHAR (QUOTE " ' CANNOT HAVE A 'THAT COMPLEMENT'. "))
     (TERPRI))
    ((NOT (OR (VTRANS HEAD)
              (DELFORTOCOMP HEAD)))
     (; Expecting the object of a transitive verb, but the verb is
        neither marked as transitive nor able to take a
        complement such as:  'I WANTED YOU TO FINISH'.)
     (PCHAR (QUOTE " UNABLE TO LOOK FOR AN OBJECT OF ' "))
     (PRINT-REG (GETR HEAD))
     (PCHAR (QUOTE " ' BECAUSE IT IS NOT TRANSITIVE AND CANNOT "))
     (PCHAR (QUOTE " TAKE A COMPLEMENT WITH A DELETED 'FOR', SUCH AS: "))
     (TERPRI)
     (PCHAR (QUOTE "      I WANT YOU TO FINISH. "))
     (TERPRI)
     (PCHAR (QUOTE " ONE OR BOTH OF THESE CONDITIONS MUST BE TRUE. "))
     (TERPRI))
    (T
     (; Blocked here for reasons other than those above.  Incomplete
        sentence, no noun phrase for the object, etc.  What was
        expected at this point is printed.)
     (PCHAR (QUOTE " TRYING TO FIND AN OBJECT OF ' "))
     (PRINT-REG (GETR HEAD))
     (PCHAR (QUOTE " ' .  WE NEED TO HAVE ONE OF THE FOLLOWING:  "))
     (TERPRI)
     (PCHAR (QUOTE "      1. A FOR/TO COMPLEMENT                 "))
     (PCHAR (QUOTE "           (IT IS POSSIBLE 'FOR ME TO...')   "))
     (PCHAR (QUOTE "      2. A THAT COMPLEMENT                   "))
     (PCHAR (QUOTE "           (I KNOW 'THAT YOU WILL...')       "))
     (PCHAR (QUOTE "           (I KNOW 'YOU WILL...')            "))
     (PCHAR (QUOTE "      3. A NOUN PHRASE - BEGINNING HERE, OR  "))
     (PCHAR (QUOTE "           HELD FROM EARLIER IN THE SENTENCE "))
     (PCHAR (QUOTE "           (I PARSED 'THE SENTENCE'.)        "))
     (TERPRI)))

 (VP/OBJ
    ((AND (WRD (THAT FOR TO))
          (EQ (GetPATHR SUBJECT HEAD HEAD)
              (QUOTE IT))
          (NOT (SUBJCOMP HEAD)))
     (; Looking for the complement of an 'IT' phrase.  The HEAD verb
        must be able to take a SUBJCOMP.  e.g., It is crazy
        'for you to parse this'.)
     (PCHAR (QUOTE " THE WORD ' "))
     (PRINT-ANY-STRING LEX)
     (PCHAR (QUOTE " ' INDICATES THE BEGINNING OF A COMPLEMENT OF "))
     (PCHAR (QUOTE " THE 'IT' PHRASE, BUT THE VERB ' "))
     (PRINT-REG (GETR HEAD))
     (PCHAR (QUOTE " ' CANNOT HAVE SUCH A COMPLEMENT. "))
     (TERPRI))
    ((AND (NOT (TOCOMP HEAD))
          (OR (WRD (NOT TO))
              (AND (FMEMB (NEXTWRD)
                          (QUOTE (NOT TO)))
                   (CATS (QADV QUESPRO)))))
 (; Having a 'TO' or 'NOT' here indicates the beginning of a
        modifying phrase that acts as the object.  I told John
        'to read the book' or We will ask her 'not to reveal
        the secret'.  However, the HEAD verb will not allow
        such a complement.)
 (PCHAR (QUOTE " THE ' "))
 (COND ((WRD (NOT TO))
        (PRINT-ANY-STRING LEX))
       (T (PRINT-ANY-STRING LEX)
          (PRINT-ANY-STRING NEXTWRD)))
 (PCHAR (QUOTE " ' SIGNALS THE BEGINNING OF A COMPLEMENT, "))
 (PCHAR (QUOTE " AS IN: "))
 (TERPRI)
 (PCHAR (QUOTE " I ASKED YOU 'TO READ THE BOOK'. "))
 (PCHAR (QUOTE " SHE TOLD US 'HOW TO DO THAT'. "))
 (TERPRI)
 (PCHAR (QUOTE " HOWEVER, THE VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' CANNOT HAVE SUCH A COMPLEMENT. "))
 (TERPRI))
((AND (TOCOMP HEAD)
      (OR (WRD (NOT TO))
          (AND (FMEMB (NEXTWRD)
                      (QUOTE (NOT TO)))
               (CATS (QADV QUESPRO)))))
 (; As above, except the verb was marked as a TOCOMP.  The push
        was successful and we returned back.  However, one of
        two aborts ended the processing of this path.)
 (COND ((NOT (ASSIGNQ * HEAD TOCOMP VERBCASES))
        (PCHAR (QUOTE " THE 'TO-COMPLEMENT' PHRASE ' "))
        (PRINT-ANY-STRING *)
        (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD WHEN USED "))
        (PCHAR (QUOTE " WITH THE VERB ' "))
        (PRINT-REG (GETR HEAD))
        (PCHAR (QUOTE " ' . ")))
       (T
        (PCHAR (QUOTE " THE NOUN PHRASE ' "))
        (PRINT-REG (GETR OBJ?))
        (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS AN INDIRECT "))
        (PCHAR (QUOTE " OBJECT OF THE VERB ' "))
        (PRINT-REG (GETR HEAD))
        (PCHAR (QUOTE " ' . "))))
 (TERPRI))
((AND (CAT PREP)
      (NOT (VPARTICLE HEAD *)))
 (; A preposition here must be with a verb that is marked as taking
        particles as an "auxiliary" part of the verb, as in
        'START UP', 'CLEAN UP', etc.)
 (PCHAR (QUOTE " A PREPOSITION, SUCH AS ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ', AT THIS POINT, SHOULD BE AN AUXILIARY "))
 (PCHAR (QUOTE " PART OF THE VERB, AS IN: "))
 (TERPRI)
 (PCHAR (QUOTE "      ...'CLEAN UP' YOUR ROOM...          "))
 (PCHAR (QUOTE "      ...'START UP' THE CAR...            "))
 (TERPRI)
```

```
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE ", HOWEVER, CANNOT HAVE SUCH AN AUXILIARY  "))
  (PCHAR (QUOTE " PREPOSITION. "))
  (TERPRI))
 ((AND (PossibleNP? T)
       (NOT (TAKEINDOBJ HEAD)))
  (; The noun phrase beginning here is the object and what is in
         register OBJ? is the indirect object if the verb can take
         an indirect object.  This verb is not marked as taking an
         indirect object.)
  (PCHAR (QUOTE " THE NOUN PHRASE BEGINNING HERE IS EXPECTED "))
  (PCHAR (QUOTE " TO BE THE OBJECT, WHILE ' "))
  (PRINT-REG (GETR OBJ?))
  (PCHAR (QUOTE " ' IS EXPECTED TO BE THE INDIRECT OBJECT. "))
  (PCHAR (QUOTE " HOWEVER, THE VERB ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' CANNOT HAVE AN INDIRECT OBJECT. "))
  (TERPRI))
 ((AND (TAKEINDOBJ HEAD)
       (PossibleNP? T))
  (; As above, except that the verb was marked as taking an
         indirect object, the noun phrase was successfully
         pushed for, but one of two aborts ended the processing
         of this path.)
  (COND ((NOT (ASSIGNQ * HEAD OBJECT VERBCASES))
         (PCHAR (QUOTE " THE NOUN PHRASE ' "))
         (PRINT-ANY-STRING *)
         (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE OBJECT "))
         (PCHAR (QUOTE " OF THE VERB ' "))
         (PRINT-REG (GETR HEAD))
         (PCHAR (QUOTE " ' . ")))
        (T
         (PCHAR (QUOTE " THE NOUN PHRASE ' "))
         (PRINT-REG (GETR OBJ?))
         (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE INDIRECT "))
         (PCHAR (QUOTE " OBJECT OF THE VERB ' "))
         (PRINT-REG (GETR HEAD))
         (PCHAR (QUOTE " ' . "))))
  (TERPRI))
 ((AND (GETR OBJ?)
       (NOT (ASSIGNQ OBJ? HEAD OBJECT VERBCASES)))
  (; A simple case of the object being in OBJ?, except that it does
         not check semantically with the verb.)
  (PCHAR (QUOTE " THE NOUN PHRASE ' "))
  (PRINT-REG (GETR OBJ?))
  (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE OBJECT OF "))
  (PCHAR (QUOTE " THE VERB ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' . "))
  (TERPRI))
 (T
  (; Blocking at this state due to any reason other than those
         handled above causes a list of what was expected at this
         state to be printed.)
  (PCHAR (QUOTE " ONE OF THE FOLLOWING WAS EXPECTED. "))
  (TERPRI)
```

```
    (PCHAR (QUOTE "  1. A SUBJECT COMPLEMENT (OF A PHRASE     "))
    (PCHAR (QUOTE "        STARTING WITH 'IT ) BEGINNING      "))
    (PCHAR (QUOTE "        WITH 'FOR', 'THAT', OR 'TO'        "))
    (PCHAR (QUOTE "        (IT IS SILLY 'FOR US TO...')       "))
    (PCHAR (QUOTE "  2. A 'TO' COMPLEMENT                     "))
    (PCHAR (QUOTE "        (I TOLD YOU 'TO READ...')          "))
    (PCHAR (QUOTE "  3. A PREPOSITION AS PART OF THE VERB     "))
    (PCHAR (QUOTE "        (HE WILL START 'UP'...)            "))
    (PCHAR (QUOTE "  4. A NOUN PHRASE ACTING AS THE OBJECT    "))
    (PCHAR (QUOTE "        (I GAVE YOU 'THE...')              "))
    (TERPRI)))

(VP/THAT?
   ((AND (OR (VTRANS HEAD)
             (DELFORTOCOMP HEAD))
         (PossibleNP? T))
    (; A transitive verb's object is pushed for (NP-noun phrase)
       and successfully found, but an abort was executed because
       the phrase cannot be an object.)
    (PCHAR (QUOTE " THE NOUN PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' CANNOT BE AN OBJECT OF ANY VERB. "))
    (TERPRI))
   (T
    (; The embedded sentence was pushed for and successfully
       found, but it does not check semantically as being a
       valid complement to the verb of the main clause.)
    (PCHAR (QUOTE " THE SENTENCE: "))
    (TERPRI)
    (PRINT-ANY-STRING *)
    (TERPRI)
    (PCHAR (QUOTE " CANNOT BE UNDERSTOOD AS A 'THAT COMPLEMENT' "))
    (PCHAR (QUOTE " PHRASE OF THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI)))

(VP/HEAD/NP
   ((NOT (OR (DELFORTOCOMP HEAD)
             (DELTOCOMP HEAD)
             (VTRANS HEAD)))
    (; The HEAD verb is not marked in any one of the three ways.
       Each of the three arcs from this state has one of the
       conditions on it.  Therefore, we are blocked.)
    (PCHAR (QUOTE " PARSING CANNOT PROCEED BECAUSE THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' IS NOT SUFFICIENTLY DESCRIBED IN THE "))
    (PCHAR (QUOTE " DICTIONARY.  WE ARE EXPECTING A COMPLEMENT "))
    (PCHAR (QUOTE " (DELFORTOCOMP OR DELTOCOMP) OR THE OBJECT "))
    (PCHAR (QUOTE " OF A TRANSITIVE VERB (VTRANS) AT THIS POINT. "))
    (TERPRI)))
```

```
(S/S/RESUME
   (T
    (; Trying to resume processing to complete a noun phrase, as
           in:  'What man did I meet in England who knew Peter?'
           The 'who' signals that the question is really not
           finished yet, and that there are more constituents to
           the NP 'What man'.)
    (PCHAR (QUOTE " CANNOT RESUME PROCESSING OF THE PHRASE ' "))
    (PRINT-REG (GETR ?))
    (PCHAR (QUOTE " ' TO PICK UP THE REST OF ITS CONSTITUENTS. "))
    (TERPRI)))


(S/S
   ((AND (WRD (FOR TO))
         (GETKEY (QUOTE FORTOCOMP)
                 HEAD VERBCASES))
    (; The FOR/TO complement for HEAD was pushed for, successfully
           found, but is semantically invalid.)
    (PCHAR (QUOTE " THE PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS A COMPLEMENT "))
    (PCHAR (QUOTE " OF THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   ((AND (NOT (CAT PREP))
         (PossibleLOCATION? T))
    (; A location phrase was pushed for, successfully found, but is
           not semantically valid with the main verb in HEAD)
    (PCHAR (QUOTE " THE 'LOCATION' PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD IN A SENTENCE "))
    (PCHAR (QUOTE " WHERE THE MAIN VERB IS ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   ((AND (NULLR LASTPUNCT)
         (GETKEY (QUOTE <TIME>)
                 HEAD VERBCASES)
         (PossibleNP?))
    (; A 'time noun phrase' (TIMENP) was pushed for, found, but is
           not semantically valid.)
    (PCHAR (QUOTE " THE 'TIME' NOUN PHRASE: "))
    (TERPRI)
    (PRINT-ANY-STRING *)
    (TERPRI)
    (PCHAR (QUOTE " CANNOT BE UNDERSTOOD WITH THE VERB ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   (T
    (; There are many more ways in which a parse can block here.
           Some would be caused by a very unusual input string, others
           would be unable to be detected through simple conditions.
           If none of the above conditions are the case, a list of
```

```
          what is expected at this point is printed.)
   (PCHAR (QUOTE " AFTER HAVING COMPLETED A MAIN CLAUSE, ONE OF  "))
   (PCHAR (QUOTE "            THE FOLLOWING WAS EXPECTED.         "))
   (TERPRI)
   (PCHAR (QUOTE " 1. A COMPLEMENT BEGINNING WITH 'FOR' OR 'TO'  "))
   (PCHAR (QUOTE "        (VERB MUST HAVE 'FORTOCOMP' FEATURE)   "))
   (PCHAR (QUOTE "        I WALKED 'FOR THE FUN OF IT'. "))
   (TERPRI)
   (PCHAR (QUOTE " 2. AN ADVERB                                  "))
   (PCHAR (QUOTE "        I WALKED 'QUICKLY'. "))
   (TERPRI)
   (PCHAR (QUOTE " 3. A PREPOSITIONAL PHRASE                     "))
   (PCHAR (QUOTE "        I WALKED 'WITH MY FRIEND'. "))
   (TERPRI)
   (PCHAR (QUOTE " 4. A 'LOCATION' PHRASE                        "))
   (PCHAR (QUOTE "        (VERB MUST HAVE '<LOCATION>' RULE)     "))
   (PCHAR (QUOTE "        I WALKED 'WHERE NO MAN HAS BEFORE'. "))
   (TERPRI)
   (PCHAR (QUOTE " 5. A 'TIME' PHRASE                            "))
   (PCHAR (QUOTE "        (VERB MUST HAVE '<TIME>' RULE)         "))
   (PCHAR (QUOTE "        I WALKED 'THE OTHER DAY'. "))
   (TERPRI)
   (PCHAR (QUOTE " 6. AN ADJUNCT PHRASE                          "))
   (PCHAR (QUOTE "        (VERB MUST HAVE APPROPRIATE ADJUNCT RULE "))
   (PCHAR (QUOTE "        I WALKED, 'SINGING ALL THE WAY'. "))
   (TERPRI)
   (TERPRI)
   (PCHAR (QUOTE " IF THE INPUT STRING DOES SATISFY ONE "))
   (PCHAR (QUOTE " OF THE ABOVE CASES, THE SEMANTIC RULES "))
   (PCHAR (QUOTE " THAT WERE INVOKED EITHER DO NOT EXIST OR "))
   (PCHAR (QUOTE " WERE APPLIED AND FAILED; I.E., THE "))
   (PCHAR (QUOTE " CONSTITUENT THAT WAS FOUND DOES NOT "))
   (PCHAR (QUOTE " 'MAKE SENSE' USING THE PRESENT LEXICON. "))
   (TERPRI)))


(S/S/PP
  ((AND (EQ (** PREP)
            (QUOTE TO))
        (TAKEINDOBJ HEAD)
        (NOT (ASSIGNQ (** POBJ)
                      HEAD INDOBJ VERBCASES)))
   (; The head of the prepositional phrase is 'TO'.  We expect the
          object of the preposition to be the indirect object of
          the full sentence, but it dos not check semantically.)
   (PCHAR (QUOTE " THE OBJECT OF THE 'TO' PREPOSITIONAL PHRASE, ' "))
   (PRINT-ANY-STRING (** POBJ))
   (PCHAR (QUOTE " ', CANNOT BE UNDERSTOOD AS THE INDIRECT "))
   (PCHAR (QUOTE " OBJECT OF THE VERB ' "))
   (PRINT-REG (GETR HEAD))
   (PCHAR (QUOTE " ' . "))
   (TERPRI))
  ((AND (EQ (** PREP)
            (QUOTE TO))
        (NOT (TAKEINDOBJ HEAD)))
   (; The preposition is 'TO' and the situation as described in the
         previous comment is expected; however, in this case, the
         HEAD verb cannot even take an indirect object.)
  (PCHAR (QUOTE " THE OBJECT OF THE 'TO' PREPOSITIONAL PHRASE, ' "))
  (PRINT-ANY-STRING (** POBJ))
  (PCHAR (QUOTE " ', IS EXPECTED TO BE THE INDIRECT OBJECT "))
  (PCHAR (QUOTE " OF THE VERB.  HOWEVER, ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' CANNOT TAKE AN INDIRECT OBJECT. "))
  (TERPRI))
 ((AND (EQ (** PREP)
           (QUOTE BY))
       (NOT (PASSIVE? ASPECT))
       (NOT (EQREG STYPE POSS-ING)))
  (; In the case where the preposition is 'BY', the object of the
         preposition is expected to be the subject of the sentence.
         The sentence is passive or a "possession-ing" as in: 'It
         was thrown by John' or 'John's winning was enjoyed by
         everyone'.  This message and the next three messages
         handle problems in interpreting the object of 'BY'.
         Here, we have neither passive nor "possession-ing".)
  (PCHAR (QUOTE " THE OBJECT OF THE 'BY' PREPOSITIONAL PHRASE, ' "))
  (PRINT-ANY-STRING (** POBJ))
  (PCHAR (QUOTE " ', IS EXPECTED TO BE THE SUBJECT OF THE        "))
  (PCHAR (QUOTE " SENTENCE.  THERE MUST EITHER BE A PASSIVE      "))
  (PCHAR (QUOTE " CONSTRUCTION OR A POSSESSION-'ING'             "))
  (PCHAR (QUOTE " CONSTRUCTION AS IN:                            "))
  (TERPRI)
  (PCHAR (QUOTE "       IT IS PARSED BY 'ME'.                    "))
  (PCHAR (QUOTE "       FRED'S WINNING WAS ENJOYED BY 'EVERYONE'. "))
  (TERPRI)
  (PCHAR (QUOTE " NEITHER IS THE CASE HERE. "))
  (TERPRI))
 ((AND (EQ (** PREP)
           (QUOTE BY))
       (PASSIVE? ASPECT))
  (; We have the passive, but the object of 'BY' does not check
         semantically as being the subject of the verb in HEAD.)
  (PCHAR (QUOTE " IN A PASSIVE CONSTRUCTION, THE OBJECT OF THE "))
  (PCHAR (QUOTE " 'BY' PREPOSITIONAL PHRASE IS EXPECTED TO BE THE "))
  (PCHAR (QUOTE " SUBJECT OF THE SENTENCE.  HOWEVER, THE WORD ' "))
  (PRINT-ANY-STRING (** POBJ))
  (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE SUBJECT WITH THE "))
  (PCHAR (QUOTE " VERB ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' . "))
  (TERPRI))
 ((AND (EQ (** PREP)
           (QUOTE BY))
       (EQREG STYPE POSS-ING)
       (NOT (ASSIGNQ FIRSTNP HEAD OBJECT VERBCASES)))
  (; In a 'POSS-ING' sentence, the object of 'BY' is the sentence
         subject and the FIRSTNP of the sentence is the sentence
         object.  Here the FIRSTNP does not check semantically as
         the object with the verb in HEAD.)
  (PCHAR (QUOTE " IN A POSSESSION-'ING' SENTENCE SUCH AS THIS, "))
  (PCHAR (QUOTE " THE FIRST NOUN PHRASE IS EXPECTED TO BE THE  "))
  (PCHAR (QUOTE " OBJECT AND THE PREPOSITIONAL OBJECT IS "))
  (PCHAR (QUOTE " EXPECTED TO BE THE SUBJECT. "))
  (TERPRI)
  (PCHAR (QUOTE " HOWEVER, THE NOUN PHRASE ' "))
  (PRINT-REG (GETR FIRSTNP))
  (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE OBJECT OF THE "))
  (PCHAR (QUOTE " VERB ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' . "))
  (TERPRI)
 ((EQ (** PREP)
      (QUOTE BY))
  (; The only other way to block with 'BY' is as in the case above,
         except the FIRSTNP as the OBJECT checks out, but the
         prepositional object as the SUBJECT doesn't.)
  (PCHAR (QUOTE " IN A POSSESSION-'ING' SENTENCE SUCH AS THIS, "))
  (PCHAR (QUOTE " THE FIRST NOUN PHRASE IS EXPECTED TO BE THE "))
  (PCHAR (QUOTE " OBJECT AND THE PREPOSITIONAL OBJECT IS EXPECTED "))
  (PCHAR (QUOTE " TO BE THE SUBJECT. "))
  (TERPRI)
  (PCHAR (QUOTE " THE NOUN PHRASE ' "))
  (PRINT-REG (GETR FIRSTNP))
  (PCHAR (QUOTE " ' IS ABLE TO BE UNDERSTOOD AS THE OBJECT, BUT ' "))
  (PRINT-ANY-STRING (** POBJ))
  (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE SUBJECT WITH THE "))
  (PCHAR (QUOTE " VERB ' "))
  (PRINT-REG (GETR HEAD))
  (PCHAR (QUOTE " ' . "))
  (TERPRI))
 ((AND (EQ (** PREP)
           (QUOTE OF))
       (NOT (EQREG STYPE POSS-ING)))
  (; When the head of the prepositional phrase is 'OF', we expect
         to have a possession-'ing' sentence.  If the verb cannot
         have an 'OFSUBJECT', then the FIRSTNP (if there is one)
         is the SUBJECT and the prepositional object is the OBJECT;
         otherwise, there should not be a FIRSTNP and the
         prepositional object is the SUBJECT.  This first 'OF'
         condition handles not having a possession-'ing' sentence.
         The next three handle aborts due to the OBJECT and/or
         SUBJECT not checking semantically.)
  (PCHAR (QUOTE " A PREPOSITIONAL PHRASE STARTING WITH 'OF' AFTER "))
  (PCHAR (QUOTE " THE END OF THE MAIN CLAUSE CAN ONLY BE "))
  (PCHAR (QUOTE " UNDERSTOOD IN A POSSESSION-'ING' SENTENCE, "))
  (PCHAR (QUOTE " SUCH AS: "))
  (TERPRI)
  (PCHAR (QUOTE "       ...JOHN'S SENDING OF THE MESSAGE...       "))
  (TERPRI))
 ((AND (EQ (** PREP)
           (QUOTE OF))
       (NOT (OFSUBJECT HEAD))
       (GETR FIRSTNP)
       (NOT (ASSIGNQ FIRSTNP HEAD SUBJECT VERBCASES)))
  (; FIRSTNP fails as SUBJECT.)
  (PCHAR (QUOTE " IN A POSSESSION-'ING' SENTENCE SUCH AS THIS, "))
  (PCHAR (QUOTE " THE 'OF' PREPOSITIONAL OBJECT IS THE OBJECT OF "))
 (PCHAR (QUOTE " THE SENTENCE AND THE PHRASE ' "))
 (PRINT-REG (GETR FIRSTNP))
 (PCHAR (QUOTE " ' IS EXPECTED TO BE THE SUBJECT. "))
 (TERPRI)
 (PCHAR (QUOTE " HOWEVER, IT CANNOT BE UNDERSTOOD AS THE "))
 (PCHAR (QUOTE " SUBJECT WITH THE VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
((AND (EQ (** PREP)
          (QUOTE OF))
      (NOT (OFSUBJECT HEAD)))
 (; Prepositional Object fails as OBJECT.)
 (PCHAR (QUOTE " IN A POSSESSION-'ING' SENTENCE SUCH AS THIS, "))
 (PCHAR (QUOTE " THE 'OF' PREPOSITIONAL OBJECT IS EXPECTED "))
 (PCHAR (QUOTE " TO BE THE OBJECT OF THE VERB. "))
 (TERPRI)
 (PCHAR (QUOTE " HOWEVER, ' "))
 (PRINT-ANY-STRING (** POBJ))
 (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE OBJECT WITH THE "))
 (PCHAR (QUOTE " VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
((EQ (** PREP)
     (QUOTE OF))
 (; The only other way 'OF' can block is if the prepositional
        object should be the SUBJECT, but it doesn't check out.)
 (PCHAR (QUOTE " THE OBJECT OF 'OF' IS EXPECTED TO BE "))
 (PCHAR (QUOTE " THE SUBJECT OF THE SENTENCE, BUT ' "))
 (PRINT-ANY-STRING (** POBJ))
 (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS THE SUBJECT WITH ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
(T
 (; If none of the above, the prepositional phrase began with
        a preposition other than 'TO', 'BY', or 'OF', or it began
        with one of those but did not need to be the SUBJECT or
        OBJECT.  In either case, a general prepositional phrase
        semantic check failed.)
 (PCHAR (QUOTE " THE PREPOSITIONAL PHRASE BEGINNING WITH ' "))
 (PRINT-ANY-STRING (** PREP))
 (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD WITH THE VERB ' "))
 (PRINT-REG (GETR HEAD))
 (PCHAR (QUOTE " ' . "))
 (TERPRI)))
```

```
(NP/PRO
   (T
    (; We can block here only if the pronoun cannot be understood
           when used with the head noun and any other modifiers of
           the head noun that there may be.  The noun phrase is
           believed to be ending with the pronoun.)
    (PCHAR (QUOTE " THE PRONOUN ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' IS BELIEVED TO END THE NOUN PHRASE.  THE "))
    (PCHAR (QUOTE " NOUN PHRASE IS UNABLE TO BE UNDERSTOOD DUE "))
    (PCHAR (QUOTE " TO INCOMPLETENESS OR INCONSISTENCY OF THE "))
    (PCHAR (QUOTE " MODIFIERS. "))
    (TERPRI)))

(DET/
   (T
    (; We can block here if 'NOT' was found and state DET/ was
           re-entered; we are looking for (but not finding) a
           determiner or an adverb followed by a comparative.)
    (PCHAR (QUOTE " THE WORD 'NOT' AS PART OF A DETERMINER PHRASE "))
    (PCHAR (QUOTE " REQUIRES A DETERMINER, OR AN ADVERB FOLLOWED "))
    (PCHAR (QUOTE " BY A COMPARATIVE, AFTER IT; AS IN: "))
    (TERPRI)
    (PCHAR (QUOTE "    NOT 'THE' LEAST OF IT IS...              "))
    (PCHAR (QUOTE "    NOT 'REALLY MORE THAN' THREE BOYS WERE... "))
    (TERPRI)))

(BASENP/NotVorADJ
   ((AND (CATS (NPR N))
         (GETR ADVS))
    (; We can have a noun or proper noun only if we have already
           used the adverbs with the adjective or participle they
           modify.  This allows 'QUICKLY RUNNING' FRED and 'VERY
           UGLY' DOG, but not 'QUICKLY FRED' or 'VERY DOG'.)
    (PCHAR (QUOTE " THE NOUN ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' IS UNABLE TO BE UNDERSTOOD HERE BECAUSE "))
    (PCHAR (QUOTE " THE ADVERB(S) ' "))
    (PRINT-REG (GETR ADVS))
    (PCHAR (QUOTE " ' CANNOT BE PART OF THE NOUN PHRASE UNLESS "))
    (PCHAR (QUOTE " IT IS FOLLOWED BY A PARTICIPLE OR AN "))
    (PCHAR (QUOTE " ADJECTIVE, AS IN: "))
    (TERPRI)
    (PCHAR (QUOTE "       ...'QUICKLY MELTING' BUTTER...      "))
    (PCHAR (QUOTE "       ...'VERY POOR' PERSON...            "))
    (TERPRI))
   ((AND (CAT YEAR)
         (GETR ADVS))
    (; As above, except that we have a year instead of a noun.)
    (PCHAR (QUOTE " THE YEAR ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD HERE BECAUSE A "))
    (PCHAR (QUOTE " PARTICIPLE OR AN ADJECTIVE IS EXPECTED AFTER "))
    (PCHAR (QUOTE " THE ADVERB(S) ' "))
```

```
 (PRINT-REG (GETR ADVS))
 (PCHAR (QUOTE " ', AS IN: "))
 (TERPRI)
 (PCHAR (QUOTE "        ...'VERY EXCITING' 1980...          "))
 (PCHAR (QUOTE "        ...'POLITICALLY WILD' 1974...       "))
 (TERPRI))
((AND (CAT MONTH)
      (GETR DET?))
 (; For the month to be the beginning of a date, there should be
        no determiners.)
 (PCHAR (QUOTE " EXPECTING THE MONTH ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " TO BE THE BEGINNING OF A DATE.  HOWEVER, "))
 (PCHAR (QUOTE " THERE CANNOT BE ANY DETERMINERS BEFORE "))
 (PCHAR (QUOTE " IT.  HERE WE HAVE ' "))
 (PRINT-REG (GETR DET?))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
((AND (WRD (AND ,))
      (EQ (CAAR (GETR PREMODS))
          (QUOTE NOUN)))
 (; When using 'AND' or ',' to conjoin something to a noun, the
        something to be conjoined must also be a noun.)
 (PCHAR (QUOTE " ONLY ANOTHER NOUN CAN BE CONJOINED "))
 (PCHAR (QUOTE " WITH THE NOUN ' "))
 (PRINT-REG (GETR PREMODS))
 (PCHAR (QUOTE " ', AS IN: "))
 (TERPRI)
 (PCHAR (QUOTE "      THE FIRST ROBIN, FLOWER AND LEAF...    "))
 (TERPRI))
((AND (WRD (AND ,))
      (SELECTQ (CAAR (GETR PREMODS))
               ((PRESPART PASTPART ADJ) T)
               NIL))
 (; When using 'AND' or ',' to conjoin something to a present
        participle, past participle, or adjective, the
        something to be conjoined must be a verb or an
        adjective.)
 (PCHAR (QUOTE " A VERB OR ADJECTIVE IS EXPECTED AFTER THE ' "))
 (PRINT-ANY-STRING LEX)
 (PCHAR (QUOTE " ' AS SOMETHING TO BE JOINED TO ' "))
 (PRINT-REG (GETR PREMODS))
 (PCHAR (QUOTE " ' . "))
 (TERPRI))
((WRD (AND ,))
 (; If blocked at 'AND' or ',' for any other reason, attempting
        an illegal conjunction.)
 (PCHAR (QUOTE " UNABLE TO FORM THE CONJUNCTION. "))
 (PRINT-REG (GETR PREMODS))
 (PCHAR (QUOTE " CANNOT HAVE ANY JOINED PHRASES. "))
 (TERPRI))
((CAT V)
 (; We should not have a verb here.  Valid ellipsis such as
        'THE LARGEST WAS...' is handled in NP/PartitiveDET.
        Here we would have 'THE WAS...'.  There is also the
        possibility of a present participle or past participle
```

```
          which was not marked as being such.)
    (PCHAR (QUOTE " A VERB CANNOT BE UNDERSTOOD HERE. "))
    (PCHAR (QUOTE " THE NOUN PHRASE WAS BEING CONSTRUCTED; "))
    (PCHAR (QUOTE " ANOTHER CONSTITUENT TO BE ADDED TO IT "))
    (PCHAR (QUOTE " WAS EXPECTED.  PERHAPS YOU ENTERED AN "))
    (PCHAR (QUOTE " INVALID ELLIPSIS , "))
    (PCHAR (QUOTE " OR THIS VERB IS A PRESENT PARTICIPLE "))
    (PCHAR (QUOTE " OR A PAST PARTICIPLE BUT IS NOT MARKED "))
    (PCHAR (QUOTE " AS SUCH IN THE DICTIONARY. "))
    (TERPRI)))


(BASENP/HEAD?
   ((AND (WRD 'S)
         (NOT (AND (PLURAL'S (CAR (GETR PREMODS)))
                   (DET/NUMBERCHECK DET? (QUOTE PLURAL) HEAD))))
    (; The 'S was used to pluralize a noun that cannot be made
          plural in that way.)
    (PCHAR (QUOTE " THE NOUN ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' CANNOT BE MADE PLURAL BY ADDING 'S. "))
    (TERPRI))
   ((NOT (OR (WRD 'S)
             (DET/NUMBERCHECK DET? NIL HEAD)))
    (; The determiner and the head noun disagree in number.)
    (PCHAR (QUOTE " THE DETERMINER ' "))
    (PRINT-REG (GETR DET?))
    (PCHAR (QUOTE " ' AND THE HEAD NOUN ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' DO NOT AGREE IN NUMBER (SINGULAR/PLURAL). "))
    (TERPRI))
   (T
    (; If blocked here not because of any of the above, then the
          semantic check on the base noun phrase (BASENP) using
          the head noun (HEAD), the determiner (DET?) and the
          premodifiers (PREMODS) was not successful.)
    (PCHAR (QUOTE " THE NOUN PHRASE CONSTRUCTED SO FAR IS "))
    (PCHAR (QUOTE " NOT ABLE TO BE UNDERSTOOD. "))
    (TERPRI)
    (PCHAR (QUOTE " THE DETERMINER: "))
    (TERPRI)
    (PRINT-REG (GETR DET?))
    (TERPRI)
    (PCHAR (QUOTE " AND THE PREMODIFIER: "))
    (TERPRI)
    (PRINT-REG (GETR PREMODS))
    (TERPRI)
    (PCHAR (QUOTE " ARE NOT ABLE TO BE UNDERSTOOD AS BEING "))
    (PCHAR (QUOTE " CONSISTENT WITH: "))
    (PRINT-REG (GETR HEAD))
    (TERPRI)))
```

```
(BASENP/SPLIT
   (T
    (; The parse will block here due to disagreement in number
           between the determiner and the noun or because the
           semantic check of the base noun phrase failed.  This
           is the same situation as with the state BASENP/HEAD?
           except that in that state, having 'S could cause a
           problem.  Since we don't have the 'S here, we can use
           BASENP/HEAD?'s condition/action pairs and the first
           pair will just always be skipped.)
    (CONDACT BASENP/HEAD?)))

(DATE/DAY&MONTH
   (T
    (; We can block here only if we had a noun phrase that was
           expected to have a FRAMETYPE of <YEAR>, but it did
           not.  In other words, instead of a year, such as '1980',
           a 'year-phrase', such as 'THE YEAR THAT...', is
           acceptable (but, only if the phrase has been
           "semantically defined" as being an instance of <YEAR>).)
    (PCHAR (QUOTE " A NOUN PHRASE DEFINING THE YEAR OF "))
    (PCHAR (QUOTE " THE DATE WAS EXPECTED.  THE PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' CANNOT BE INTERPRETED AS A YEAR. "))
    (TERPRI)))

(DATE/END
   (T
    (; The end of the date (and the noun) phrase is expected.  We
           block here if DAY, MONTH, and YEAR do not form a
           semantically valid <DATE>.)
    (PCHAR (QUOTE " THE DATE CONSTRUCTION IS BELIEVED TO BE "))
    (PCHAR (QUOTE " COMPLETED.  HOWEVER, THE COMPONENTS "))
    (TERPRI)
    (COND ((GETR DAY)
           (PCHAR (QUOTE " DAY: "))
           (PRINT-REG (GETR DAY))))
    (TERPRI)
    (COND ((GETR MONTH)
           (PCHAR (QUOTE " MONTH: "))
           (PRINT-REG (GETR MONTH))))
    (TERPRI)
    (COND ((GETR YEAR)
           (PCHAR (QUOTE " YEAR: "))
           (PRINT-REG (GETR YEAR))))
    (TERPRI)
    (PCHAR (QUOTE " ARE UNABLE TO BE UNDERSTOOD TOGETHER "))
    (PCHAR (QUOTE " AS A VALID DATE. "))
    (TERPRI)))
```

```
(NP/PartitiveDET
   (T
    (; We can block here only if a word is not classified fully
          as a NOUN, ADJ, PASTPART, etc. or if the sentence is
          not grammatical.  Whatever the case, what was expected
          is printed.)
    (PCHAR (QUOTE " AFTER A PARTITIVE DETERMINER SUCH AS ' "))
    (PRINT-REG (GETR PARTITIVE/DET))
    (PCHAR (QUOTE " ', ONE OF THE FOLLOWING WAS EXPECTED. "))
    (COND ((GETR ALLDET)
          (PCHAR (QUOTE "   'THE' OR A POSSESSIVE PROUNOUN    "))
          (PCHAR (QUOTE "         (ALL 'THE' PEOPLE...)       "))
          (PCHAR (QUOTE "         (ALL 'HER' PAPERS...)       "))))
    (PCHAR (QUOTE "   A NOUN                                       "))
    (PCHAR (QUOTE "        (THE LARGEST 'TREE'...)                 "))
    (PCHAR (QUOTE "   A PRESENT OR PAST PARTICIPLE                 "))
    (PCHAR (QUOTE "        (THE FIRST 'DEFEATED' ARMY...)          "))
    (PCHAR (QUOTE "   AN ADJECTIVE                                 "))
    (PCHAR (QUOTE "        (TWO 'QUICK' FOXES...)                  "))
    (PCHAR (QUOTE "   AN ADVERB                                    "))
    (PCHAR (QUOTE "        (THE FIRST 'SLOWLY' MOVING VEHICLE...) "))
    (PCHAR (QUOTE "   AN 'OF' PREPOSITIONAL PHRASE                 "))
    (PCHAR (QUOTE "        (THE LAST THREE 'OF' THE MEN...)        "))
    (TERPRI)
    (PCHAR (QUOTE " IF NONE OF THE ABOVE ARE TRUE, THEN "))
    (PCHAR (QUOTE " AN ELLIPTICAL REFERENCE TO A PREVIOUS "))
    (PCHAR (QUOTE " NOUN (ELLIPTICAL ANAPHORA) IS EXPECTED. "))
    (PCHAR (QUOTE " (THE FIRST WAS...    OR     THREE ARE...) "))
    (PCHAR (QUOTE " HOWEVER, THERE IS NO EVIDENCE TO INDICATE "))
    (PCHAR (QUOTE " THAT THIS IS THE CASE HERE EITHER. "))
    (TERPRI)))

(NPLIST/
   (T
    (; We can block here only if the newly found noun phrase cannot
          be compatible with the noun phrase(s) it is conjoined to.)
    (PCHAR (QUOTE " THE NOUN PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' IS NOT COMPATIBLE WITH THE NOUN PHRASE(S) "))
    (PCHAR (QUOTE " THAT IT IS CONJOINED TO. "))
    (TERPRI)))

(NPLIST/NP
   (T
    (; We can block here only if the conjoining of the noun phrases
          was not correct.)
    (PCHAR (QUOTE " CONJOINED NOUN PHRASES WERE EXPECTED, "))
    (PCHAR (QUOTE " BUT THE STRUCTURE OF THE SENTENCE "))
    (PCHAR (QUOTE " AT THIS POINT MAKES SUCH AN INTEPRETATION "))
    (PCHAR (QUOTE " IMPOSSIBLE. "))
    (TERPRI)))
```

```
(NP/INTID
   ((NOT (ASSIGNQ INTIDS HEAD IDENTIFIER POSTMODCASES))
    (; We will block at this state if the integer identifier(s) are
          not able to check semantically with the HEAD noun (this
          condition) or if they do check alone with the HEAD, but
          do not check along with all of the other modifiers of
          HEAD (the following condition).)
    (PCHAR (QUOTE " THE INTEGER(S) "))
    (PRINT-REG (GETR INTIDS))
    (PCHAR (QUOTE " CANNOT BE UNDERSTOOD AS IDENTIFYING ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   (T
    (PCHAR (QUOTE " ALTHOUGH THE INTEGER IDENTIFIER(S) "))
    (PRINT-REG (GETR INTIDS))
    (PCHAR (QUOTE " ARE UNDERSTOOD AS IDENTIFIERS OF ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ', THE OTHER MODIFIERS OF IT "))
    (PCHAR (QUOTE " ARE NOT CONSISTENT WITH THIS "))
    (PCHAR (QUOTE " INTERPRETATION. "))
    (TERPRI)))

(NP/APPOSITE/NP
   ((NOT (WRD ,))
    (; We block here if the appositive does not end (as it started)
          with a comma, or if the appositive phrase does not check
          semantically with the HEAD noun.)
    (PCHAR (QUOTE " AN APPOSITIVE NON PHRASE, SUCH AS ' "))
    (PRINT-REG (GETR APPOSITE))
    (PCHAR (QUOTE " ', SHOULD END WITH A COMMA. "))
    (TERPRI))
   (T
    (PCHAR (QUOTE " THE APPOSITIVE NOUN PHRASE ' "))
    (PRINT-REG (GETR APPOSITE))
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS A MODIFIER OF ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI)))
```

```
(NP/POSTMODS?
   ((LISTP *)
    (; If * is a list, then its value is the prepositional phrase
           that was found at the end of the noun phrase.  We block
           here if it does not check semantically as a modifier of
           the HEAD noun.)
    (PCHAR (QUOTE " THE PREPOSITIONAL PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' CANNOT BE UNDERSTOOD AS A MODIFIER "))
    (PCHAR (QUOTE " OF THE NOUN ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   ((OR (CAT INTEGER)
        (WRD (NUMBER NUMERS)))
    (; This handles the case where (from NP/BASENP) a noun was
           numerically identified, but it is not allowed to be so
           identified (semantically), so the jump to NP/POSTMODS?
           was taken, and here we block.)
    (PCHAR (QUOTE " THE NOUN ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' CANNOT HAVE A NUMERIC IDENTIFIER. "))
    (TERPRI))
   (T
    (; The noun phrase is believed to be complete.  The semantic
           check on the consistency and completeness of the
           modifiers of the HEAD noun failed.)
    (PCHAR (QUOTE " THE NOUN PHRASE IS BELIEVED TO BE "))
    (PCHAR (QUOTE " COMPLETE.  HOWEVER, THE MODIFIERS OF ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' ARE NOT CONSISTENT; THEY ARE NOT "))
    (PCHAR (QUOTE " ALL ABLE TO BE UNDERSTOOD. "))
    (TERPRI)))


(NP/SMOD?
   (T
    (; We will block here if the clause-type postmodifier was
           found but does not check semantically.)
    (PCHAR (QUOTE " THE CLAUSE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' IS NOT ABLE TO BE UNDERSTOOD "))
    (PCHAR (QUOTE " AS A MODIFIER OF ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI)))


(NP/RESUME
   ((EQ (CAAR *)
        (QUOTE PP))
    (; The prepositional phrase does not check semantically (this
           condition) or the clause-type postmodifier does not
           check semantically (next condition).)
    (PCHAR (QUOTE " THE PREPOSITIONAL PHRASE ' "))
    (PRINT-ANY-STRING *)
    (PCHAR (QUOTE " ' IS NOT ABLE TO BE UNDERSTOOD "))
    (PCHAR (QUOTE " AS A MODIFIER OF ' "))
    (PRINT-REG (GETR HEAD))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   (T
    (CONDACT NP/SMOD?)))
```

S/PP is entered via a transition from state S/ after a prepositional phrase has been successfully pushed for and popped. If there is a "," here, it is consumed and a transition is made to state S/, otherwise the transition is made to state S/ without consuming any of the input string.

S/ADJUNCT is entered via a transition from state S/ after an adjunct has been successfully pushed for and popped. If there is a "," here, it is consumed and S/ADJUNCT is re-entered. If there is a 'THEN' and we had an IF as a binder, then the 'THEN' is consumed and a transition made to state S/. Otherwise, the transition is made to S/ with no consumption of the input string.

S/DECL is entered via a transition from S/, P/COMP, SMOD/1, SMOD/RELPP or SMOD/WHENorWHERE. In all cases, a declarative sentence is expected. If there is a 'FOR', 'TO', or 'THAT' here, a JUMP to S/SUBJCOMP? is taken. If we have had a relative clause with an explicit subject and a relative pronoun ('THE MAN WHOM THE BOY TOLD US ABOUT.'), a JUMP to S/NP is taken. Otherwise, a noun phrase is examined by a PUSH for state NP/.

Q/HOW is entered via a transition from state Q/. We had the word 'HOW' and now have an adjective or an adverb. After the appropriate register-setting actions, state S/NP is entered, the adjective or adverb having been consumed.

S/SUBJCOMP? is entered after finding a 'FOR', 'TO' or 'THAT' in state S/DECL. If the word is not 'FOR' or 'TO' state NP/ is PUSHed; if it is 'FOR', 'TO' or 'THAT', then SMOD/ is PUSHed. We are looking for the subject or subject complement.

S/NP is entered from ADJUNCT/, Q/, Q/HOW, S/, S/DECL, S/SUBJCOMP?, SMOD/1, SMOD/FOR/COMPL, SMOD/FOR/NP, SMOD/NPONLY, or SMOD/RELPP. In all cases, a verb or adverb is expected after having had the word 'TO', the word 'THERE', the word 'HOW' followed by an adjective or an adverb, a noun phrase, or a modifying noun phrase. If a verb is on the hold list, a JUMP is taken to state VP/V; if the current word is an adverb, it is consumed and S/NP re-entered; otherwise, a JUMP to S/NP/1 is made.

VP/HEAD is entered from VP/ASSIGNHEAD, VP/UNTENSEDandPASTPART, or VP/V. Either there is a passive construction and the verb has been found, the main verb can take a preposition and it was found, or a noun phrase was found to complete a 'THERE' phrase. Whatever the case, the verb is complete and objects are expected. If the head verb is intransitive or the sentence is a POSS-ING ("MY LEAVING WAS..."), then a JUMP to VP/INTRANS is made; otherwise, a JUMP to VP/GETOBJ.

VP/INTRANS is entered via a transition from VP/HEAD above. If the verb can also be transitive and there is evidence for an object (one has been found but not placed yet, the verb can take a 'FOR/TO' complement, or a NP is possibly next), then a JUMP to VP/GETOBJ is taken; otherwise, a JUMP to VP/OBJ.

NP/ is entered via a JUMP from NP/PartitiveDET or a PUSH from a number of states. 1NP/ and NP/ form the beginning state group of the noun phrase analysis. If the current word is 'BOTH' or 'EITHER', NPLIST/ is entered and the word is consumed. If a noun phrase is on the hold list, state NP/VIR or state NP/WAITVIR is JUMPED to. Otherwise, a PUSH for 1NP/ is made.

NP/VIR is entered from NP/ above. A noun phrase is on the hold list. If there is now another possible noun phrase, 1NP/ is PUSHed for. Otherwise, the held noun phrase is removed and a transition to state POPIT/ is made.

NP/WAITVIR is entered from NP/ above. A noun phrase is on the hold list and there may be another one beginning here. The held noun phrase is removed from the hold list and state POPIT/ is entered. Because of a WAIT at that arc, state 1NP/ is first PUSHed for, and, if successful before the WAIT is over, state NPLIST/NP is entered.

NP/BASENP is entered from BASENP/SPLIT, BASENP/HEAD?, NP/PartitiveDET, 1NP/, NPLIST/NP, or NP/PRO. The noun phrase may be completed. Possessives (' or 'S), integer identifiers, POSS-ING participle, or appositive phrases are looked for. If none of the above, NP/POSTMODS? is entered to find any prepositional or relative clause postmodifiers.

1NP/ is PUSHed for from NP/ above. If a year, date, string, pronoun, or determiner is here, BASENP/, NP/BASENP, NP/PRO, or DET/ is entered. Otherwise, state BASENP/ is JUMPed to.

NP/POSS is entered from NP/BASENP. The determiners and head noun have been found, and a "'" or a "'S" indicating a possessive was consumed. State DET/POSTARTS? is PUSHed to find further additions to the noun phrase - ordinals, numbers, superlatives, etc.

NP/APPOSITE is entered via a transition from NP/BASENP. The noun phrase is believed to be completed, a "," was found, and the head noun can take an appositive phrase. State NP/ is PUSHed to analyze the appositive.

NP/POSTMODS?/PP is entered from NP/RESUME or NP/POSTMODS?. In both cases, a prepositional phrase (as a noun phrase modifier) was just consumed. If there is a "," here and a preposition following that, the comma is consumed and NP/POSTMODS? entered. Otherwise, a JUMP to NP/POSTMODS? is made.

BASENP/ is entered from 1NP/, NP/PartitiveDET, or DET/POSTARTS?. In the first case, no determiners, dates, strings, or pronouns were found at the beginning of the noun phrase; the "base" of the noun phrase can be looked for. In the second case, a partitive determiner in a short form (no prepositional phrases) was found and the "base" of the noun phrase is next. In the last case, a year was found, or there are no more ordinals, numbers, superlatives, etc. - the determiner is completed and the "base" of the phrase is next. If the current word is an adjective, present participle, or past participle, BASENP/VorADJ is JUMPed to. If it is both a noun and an adjective, present participle, or past participle (such as 'GREEN', 'FRAME', etc.), then a JUMP is made to BASENP/HARD. Otherwise, BASENP/NotVorADJ is JUMPed to.

BASENP/VorADJ is entered from BASENP/ above. This transition was taken because the current word was an adjective, present participle, or past participle. The three arcs in this state each handle one of the three cases. Because there are no conditions or semantic ABORTs, this state serves to set registers; it cannot block the parse. BASENP/ is entered in the adjective and past participle cases, BASENP/HEAD? in the present participle case.

BASENP/HARD is entered via a transition from BASENP/ above. If there have been no adverbs, then the word is consumed and BASENP/HEAD? entered. Otherwise, we have the situation as described in BASENP/VorADJ above. (If there were no adverbs, we take the noun interpretation of the word.)

DATE/MONTH is entered from BASENP/NotVorADJ. There were no determiners and a month was consumed. The date may be in any of the following forms: 'MAY THE...', 'MAY 18...', 'MAY SIXTH...', or 'MAY 1980'. If the current word is 'THE', it is consumed and DATE/MONTH re-entered. If it is an integer (less than 32), the day is found and a transition taken to DATE/DAY&MONTH. If it is an ordinal (other than 'LAST'), the day is found and DATE/DAY&MONTH entered. Otherwise, a JUMP to DATE/DAY&MONTH is taken.

DET/POSTARTS? is entered from DET/, RELNP/WHOSE, or PARTITIVERELNP/POSTARTS?. In all cases, the beginning of a determiner was found ('THE', a quantity, an ordinal, etc.). This state picks up the remaining parts of the determiner. If the current word is a year, BASENP/ is JUMPed to. If it is a month and there have been no ordinals or superlatives, but there was a quantity, then DATE/DAY&MONTH is entered (it's a date). Superlatives, ordinals, and integers are consumed and DET/POSTARTS? re-entered. If there was a partitive determiner, a JUMP to NP/PartitiveDET is taken. Otherwise, the determiner must be completed, and a JUMP to BASENP/ is made.

[Note: All of the *BAKSTK* messages were written, not just those for the S, VP, and NP groups. The action (function) PRINT-EM defined below is used by some of the *BAKSTK* messages to output what is known about the main sentence.]

```
(DEFPROP
   PRINT-EM
     (LAMBDA X
         (PCHAR (QUOTE " IN THE MAIN SENTENCE: "))
         (COND ((GETR SUBJECT)
                (TERPRI)
                (PCHAR (QUOTE " SUBJECT UNDERSTOOD TO BE: "))
                (PRINT-REG (GETR SUBJECT))))
         (COND ((GETR HEAD)
                (TERPRI)
                (PCHAR (QUOTE " VERB UNDERSTOOD TO BE: "))
                (PRINT-REG (GETR HEAD))))
         (COND ((GETR OBJECT)
                (TERPRI)
                (PCHAR (QUOTE " OBJECT UNDERSTOOD TO BE: "))
                (PRINT-REG (GETR OBJECT))))
         (COND ((GETR INDOBJ)
                (TERPRI)
                (PCHAR (QUOTE " INDIRECT OBJECT UNDERSTOOD "))
                (PCHAR (QUOTE " TO BE: "))
                (PRINT-REG (GETR INDOBJ)))))
   FEXPR)

(*BAKSTK*ADJUNCT/BECAUSE
   ((AND (EQREG BINDER BECAUSE)
         (EQREG STYPE ADJUNCT))
    (; The error occurred while processing the adjunctive embedded
          sentence.)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE EMBEDDED SENTENCE "))
    (PCHAR (QUOTE " AS AN ADJUNCT TO THE MAIN SENTENCE. "))
    (PCHAR (QUOTE " EVERYTHING BEFORE THE 'BECAUSE' WAS UNDERSTOOD. "))
    (TERPRI)
    (TERPRI)
    (PRINT-EM)
    (TERPRI)))
```

```
(*BAKSTK*ADJUNCT/BINDER
   ((AND (EQREG STYPE ADJUNCT)
         (NOT (WRD THE)))
    (; The error occurred as above, except that the BINDER is not
           'BECAUSE'.)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE EMBEDDED SENTENCE "))
    (PCHAR (QUOTE " (BEGINNING WITH ' "))
    (PRINT-ANY-STRING LEX)
    (PCHAR (QUOTE " ') AS AN ADJUNCT TO THE MAIN SENTENCE. "))
    (TERPRI)
    (TERPRI)
    (PRINT-EM)
    (TERPRI)))


(*BAKSTK*S/ADJUNCT
   ((AND (EQ (GetPATHR ADJUNCT BINDER)
             (QUOTE IF))
         (WRD THEN))
    (; The error occurred while parsing the 'THEN' sentence part
           of an 'IF <sent 1> THEN <sent 2>' phrase.)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE MAIN SENTENCE "))
    (PCHAR (QUOTE " BEGINNING AFTER 'THEN' FOLLOWING "))
    (PCHAR (QUOTE " THE ADJUNCT ' "))
    (PRINT-REG (GETR ADJUNCT))
    (PCHAR (QUOTE " ' . "))
    (TERPRI))
   (T
    (; The error occurred during the processing of the main
           sentence after the adjunct (but not 'IF...THEN'...).)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE MAIN SENTENCE "))
    (PCHAR (QUOTE " FOLLOWING THE ADJUNCT ' "))
    (PRINT-REG (GETR ADJUNCT))
    (PCHAR (QUOTE " ' . "))
    (TERPRI)))


(*BAKSTK*S/PP
   (T
    (; This error occurred while processing the main sentence
           after a preposed prepositional phrase.)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE MAIN SENTENCE "))
    (PCHAR (QUOTE " FOLLOWING THE PREPOSITIONAL PHRASE. "))
    (TERPRI)))
```

```
(*BAKSTK*SMOD/RELPP
   (NOT (WRD TO))
    (; The modifying phrase began with a prepositional phrase.
           The error occurred after the prepositional/modifying
           phrase, while processing the embedded sentence.
           (THE DOG TO WHICH 'I GAVE THE BONE.', THE MAN FROM
           WHOSE WORDS 'WE ALL DRAW INSPIRATION.').)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING AN EMBEDDED SENTENCE "))
    (PCHAR (QUOTE " WITHIN THE MODIFYING PHRASE THAT "))
    (PCHAR (QUOTE " BEGAN WITH THE PREPOSITION ' "))
    (PRINT-REG (GETR HEADPREP)
    (PCHAR (QUOTE " ' . "))
    (TERPRI)))

(*BAKSTK*SMOD/WHENorWHERE
   (T
    (; This error occurred during the processing of the embedded
           sentence following the 'WHEN' or 'WHERE'.  (FRED WAS
           EMBARRASSED WHEN 'IT HAPPENED'.))
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE PHRASE BEGINNING WITH ' "))
    (PRINT-REG (GETR RELWRD))
    (PCHAR (QUOTE " ' AS AN EMBEDDED SENTENCE MODIFYING "))
    (PCHAR (QUOTE " THE MAIN SENTENCE. "))
    (TERPRI)
    (TERPRI)
    (PRINT-EM)
    (TERPRI)))

(*BAKSTK*VP/THAT?
   (T
    (; This error occurred during the processing of an embedded
           sentence following a verb that takes a 'THAT' complement
           (with or without the actual word 'THAT').)
    (PCHAR (QUOTE " THIS ERROR OCCURRED WHILE THE SYSTEM "))
    (PCHAR (QUOTE " WAS INTERPRETING THE EMBEDDED SENTENCE "))
    (PCHAR (QUOTE " AS A 'THAT' COMPLEMENT OF THE MAIN "))
    (PCHAR (QUOTE " SENTENCE. "))
    (TERPRI)
    (TERPRI)
    (PRINT-EM)
    (TERPRI)))
```