SRI International

A D-LADDER USER'S GUIDE

Technical Note 224

September 1980

By: Daniel Sagalowicz, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

Attn: Lcdr. A. J. Dietzler

Contract N00039-79-C-0118
ARPA Order No. 3175.28
SRI Project 7910

The development of the D-LADDER system has been supported by the Advanced Research Projects Agency of the Department of Defense under contract DAAG29-76-C-0012 with the U.S. Army Research Office and contracts N00039-78-C-0060 and N00039-79-C-0118 with the Naval Electronic Systems Command.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

ABSTRACT

D-LADDER (DIAMOND-based Language Access to Distributed Data with Error Recovery) is a computer system designed to provide answers to questions posed at the terminal in a subset of natural language regarding a distributed data base of naval command and control information. The system accepts natural-language questions about the data. For each question D-LADDER plans a sequence of appropriate queries to the data base management system, determines on which machines the queries are to be processed, establishes links to those machines over the ARPANET, monitors the processing of the queries and recovers from certain errors in execution, and prepares a relevant answer to the original question.

This user's guide is intended for the person who knows how to log in to the host operating system, as well as how to enter and edit a line of text. It does not explain how D-LADDER works, but rather how to use it on a demonstration basis.

CONTENTS

I INTRODUCTION

D-LADDER (Language Access to Distributed Data with Error Recovery) is a computer system designed to provide answers to questions posed at the terminal in a subset of natural language regarding a distributed data base of naval command and control information.* The system accepts natural-language questions about the data. For each question D-LADDER plans a sequence of appropriate queries to the data base management system, determines on which machines the queries are to be processed, establishes links to those machines over the ARPANET, monitors the processing of the queries and recovers from certain errors in execution, and prepares a relevant answer to the original question.

This user's guide is intended to provide a relatively brief characterization of the system's current capabilities. It presumes that a prospective user can already log in to a host computer supporting D-LADDER, and that he can type in (and, if necessary, correct) a line of text.

The D-LADDER system, operational since June 1980, is undergoing continual improvement. The concepts underlying the development and operation of such a system are described in detail in the technical literature [1] [2] [3] [4] [5]. D-LADDER is written in INTERLISP [6] and uses SRI's DIAMOND package [7] [8] for building natural-language interfaces.

D-LADDER is currently installed on SRI-KL at SRI International. It is also installed on a PDP-10 in the Advanced Command Control Architectural Testbed (ACCAT) at the Naval Ocean Systems Center. At

---

* The development of the D-LADDER system has been supported by the Defense Advanced Research Projects Agency of the Department of Defense under contracts DAAG29-76-C-0012 with the U.S. Army Research Office and the contracts N00039-78-C-0060 and N00039-79-C-0118 with the Naval Electronic Systems Command, respectively.

SRI-KL, D-LADDER runs under the TOPS-20 operating system; at ACCAT, it runs under TENEX.

The data base used by D-LADDER is described in detail elsewhere [9]. It is currently stored on a data base management system called Datacomputer [10]. Use of the D-LADDER system on a demonstration basis does not require familiarity with any of these references.

If you have any questions or comments, please direct them to the authors of this manual.*

---

* ARPANET messages can be sent to SAGALOWICZ@SRI-KL

II INTERACTING WITH D-LADDER IN ENGLISH

To run D-LADDER, simply type

D-LADDER

followed by a carriage return to the system when you are at the EXEC level.

Once you are in the D-LADDER system, all interactions are performed in a subset of English. This section will provide general guidelines about the English-language interface. Subsequent sections will describe particular classes of commands and questions.

When D-LADDER prompts you with a question number followed by a left arrow (which appears as an underscore on some terminals), type in your question, terminating it with a carriage return.* Punctuation is permitted but not required. D-LADDER accepts input in any combination of upper and lowercase. Backspacing is done by typing control-A or the delete key, depending on whether the host operating system is TENEX or TOPS-20, respectively. You may abort the processing of a question at any time by typing control-D. Further procedures for editing a line of input are described in Section 14 of the INTERLISP manual [6].

An attempt has been made to accept a wide range of English-language inputs that are relevant to both the data base and the task of command-control decision-making. The examples given in the following sections are therefore to be taken as suggested, rather than mandatory, forms. Users are encouraged to experiment with different constructions, as this is how gaps are exposed in our coverage of potential questions. Note that while all the following examples are presented in uppercase, you may use uppercase, lowercase, or a combination of the two in typing to D-LADDER.

* In general, D-LADDER will interpret any carriage return as the end of a question. If you are typing a question longer than one line, either do not type a carriage return until the end, or precede the carriage return with a blank.

III QUERYING THE DATA BASE

D-LADDER makes very strong assumptions about what you are going to type to it. In particular, it assumes that data base queries are relevant to the data base it is designed to access. Questions about kinds of information that are not contained in the data base (for example, sensor ranges) normally cannot be parsed (i.e. they cannot be recognized by D-LADDER as an instance of any pattern in the grammar). Questions about ships, places, or facts that are not in the data base will result in failures to parse or in incomplete answers. For example, the question,

HOW FAR IS THE KITTY HAWK FROM MALTA?

cannot be parsed, because Malta is not in the data base and hence not in D-LADDER's vocabulary. The question,

WHO COMMAND THE CARRIERS

will not give information about the Soviet carriers that might be retrieved, because no information on their commander is available.

Since the data bases that D-LADDER accesses are concerned with ships, ship characteristics, and ship movements, we shall first describe the ways you can refer to ships.

A. Referring to Ships

Ships can be specified by name (e.g., Pogy), class (e.g., Kitty Hawk), type (e.g., cargo freighter), or naval ship classification (e.g., SSBN). D-LADDER recognizes the names of all US Navy ships, plus those foreign and merchant ships that are in the Blue File data base.

Examples of valid ship specifications are:

THE CHARLES F. ADAMS
ETHAN ALLEN CLASS SUBMARINES
CGN
INTELLIGENCE COLLECTORS.

The specification of a ship can be modified by appending its country, kind of operation, or distinguishing feature. For example:

AMERICAN CRUISER
NUCLEAR-POWERED VESSELS

The specification can also be modified by more complex phrases and clauses, expressing comparisons of characteristics, comparisons with other ships, specifications of position, or indications of a ship's route, cargo, or casualty status. For example:

AMERICAN CARGO FREIGHTERS
LIBERIAN TANKERS HEADING FOR AMERICA

Additional types of modifications can be performed by specifying the values of particular attributes. For example:

SHIPS WITH A DOCTOR

B. Asking Questions about Ships

Many simple questions about ships seek to determine what ships satisfy a given set of restrictions. These questions correspond closely to the more complex restrictions on ships described in the previous section.

You can ask for ships of any particular class (e.g., Kitty Hawk), type (e.g., cargo freighter), or naval classification (e.g., SSBN).

Examples of simple restriction-type questions are:

FIND THE LOS ANGELES CLASS SUBMARINES
WHAT SHIPS ARE CRUISERS?
WHERE ARE THE SHIPS OF TYPE DDG.

C. Questions about Attributes of Ships

Most questions typically asked of a data base are concerned with the current values of attributes that are explicitly stored. D-LADDER provides many formats for specifying such questions. The simplest forms ask for the stored attributes. For example:

WHAT IS THE RADIO CALL SIGN OF THE FOX?
WHAT IS THE CURRENT POSITION OF THE CARRIERS
WHAT IS FUEL STATUS OF THE DESTROYERS

Many more formats permit asking in subtler ways about attributes of ships. For example:

WHERE WILL THE DUTCH CARGO FREIGHTERS GO?
WHEN WILL THE CALIFORNIA ARRIVE IN NAPLES?
TO WHAT TASK GROUP DO THE DDGS BELONG?
WHAT CLASS DOES THE HOEL BELONG TO
WHO COMMANDS THE STERETT?

IV ELLIPTICAL QUESTIONS AND COMMANDS

D-LADDER accepts not only a complete sentence, but also a sentence fragment that can be interpreted in the context of the preceding sentence. The syntactic term for this condition, in which words of a second sentence are left out but implied, is ellipsis.

When an input cannot be interpreted as a complete sentence, D-LADDER types out the message, "Trying Ellipsis:" and then verifies whether it is analogous to any contiguous string of words in the preceding sentence. If it is, the input is substituted for that string and the new sentence is printed out. D-LADDER then proceeds to carry out the resulting request. Examples of valid elliptical inputs in the context of the previous question, WHAT IS THE LENGTH OF THE SANTA INEZ include:

```
THE BEAM AND DRAFT
HOME PORT OF THE AMERICAN CARRIERS
PRINT THE NATIONALITY
KITTY HAWK
```

If, however, no analogy can be found between the new input and any substring of the preceding input, D-LADDER prints out "Ellipsis has failed," and prints an error message.

WHAT ABOUT <frag>? can also be used as an alternative to <frag> where <frag> is a sentence fragment. Thus D-LADDER will accept the sequence:

```
WHAT IS THE LENGTH OF THE FOX?
WHAT ABOUT DRAFT?
```

Elliptical fragments can also be added to the end of the previous sentence, as in the sequence:

```
WHAT ARE THE US CARRIERS?
IN THE MED?
```

V DEFINING NEW GRAMMAR AND LEXICON

This chapter contains information about how to extend the D-LADDER system. Because the D-LADDER grammar is a general grammar of English, it is unlikely that any user will want to extend the grammar.* Most of the extensions ordinary users will need to make can be accomplished by adding new words to D-LADDER. The linguistic coverage of the grammar provides for handling most of the constructions in which the new word can occur. In this sense, D-LADDER is an improvement over S-LADDER. However, this advantage is accompanied by an expense for several categories of words (e.g., adjectives, verbs, and relational nouns): augmenting the D-LADDER vocabulary requires that the user enter more sophisticated linguistic information about the words and be familiar with the underlying data base schema.

In general, three different kinds of entries are necessary to add new words to D-LADDER: (1) a lexical entry, (2) a conceptual schema entry, and (3) a data base connector. In some cases, (2) and (3) will have already been set up by previous entries. Except in such cases and for common nouns that refer to field values (e.g., oil as a value in the CARGO field), the need for (2) and (3) means that the user must know the data base organization to add new words.

The next subsection describes how to make new lexical entries. Because the kind of information needed in categories (2) and (3) depends on whether the item being added refers to an "object" (e.g., oilers) or a "predicate" (e.g., carrying or commanding), these are described separately in the following subsections.

---

* Appendix A describes the DIAMOND system and the mechanisms it includes for adding and changing grammar rules. Only users with a solid understanding of linguistics should attempt to change the grammar.

A. Creating Lexical Items

Lexical entries consist of a word, a category, and a list of attributes (attribute value pairs) for the word. Most of the attributes to be added to words are self-explanatory (or easily derived by looking at a few examples). Sample lexical entries can be examined by typing

```
!SHOWWORD(sample-word)
```

Multiword entries are essentially like single words except that the whole entry is in parentheses.* Different attributes are required for nouns, adjectives, and verbs. The entries for common nouns are straightforward; other entries are more complex. Each category will be described separately along with examples.

To add words to the lexicon either use your favorite editor or look in the appendix under section 3 (Defining and Editing Words and Categories). Functions of particular interest are:

```
WORDS.DEF
EDITWORD
ADDWORDATTR
SHOWWORD
```

If you use the DIAMOND functions you'll need to do a SAVECATEGORIES for any categories (e.g., N, ADJ) that you add words to. Word definitions do not get compiled so SAVECATEGORIES only gets called with a filename.

1. Nouns

The following attributes are required for all nouns:

CLASS -- must refer to a node in the conceptual schema

ELEMENT -- must refer to some value in the data base

TYPE -- for nouns, value is one of COMMON, MASS, PROPERN

Example:

```
(ATLANTIC (ELEMENT . ATLANTIC))
(CLASS . OCEAN-NAMES)
(TYPE . PROPERN))
```

---

* There should be far fewer of these in a D-LADDER system than in a LIFER-based system

Several additional attributes are required for relational nouns (nouns that express a relation between two kinds of entities, e.g., draft. They are used in constructions like "draft of the ship" "draft of 100 feet"). Because each field in the data base gives rise to at least one relational noun, such entries arise often and will be described in more detail.

Each field in the data base corresponds to an underlying predicate. This predicate relates an entity--which is a subject of the relation--to a property of that entity typically stored in that relation as a field value. The field name functions grammatically like a noun, but the semantics for the predicate include arguments (unlike the semantics for common nouns, which only require a CLASS specification).

The following attributes are required for relational nouns:

PREDICATE.INDICATOR -- value is a predicate node in the conceptual schema.

PATTERNS -- gives the mapping from surface form to thematic cases. The value for relational nouns is PAT.RELN.OF (this attribute is also used for adjectives and verbs).

TD.MAP -- gives the mapping from thematic form to conceptual schema cases. The two most common cases that must be given are OF and OF.EQ.

RELATIONAL -- value is either T or SCALE. SCALE is most common. It is used for words like "length" which can occur in constructions with either of their two arguments (e.g., "length of a ship" "length of five feet"). The value T is used for words like "command" ("command of an admiral" but not "command of the Lafayette").

Examples:

```
((CALL SIGN)(PREDICATE.INDICATOR . CALL-SIGNINGS)
(TD.MAP (OF.EQ . CALLSIGN)
(OF . SHIP))
(CLASS . CALLSIGNS))
(PATTERNS PAT.RELN.OF)
(RELATIONAL . SCALE)
(TYPE . COMMON))
(COMMAND (PATTERNS PAT.RELN.OF)
(RELATIONAL . T)
(TYPE . COMMON))
(PREDICATE.INDICATOR . COMMANDINGS)
(CLASS . UNITS)
(TD.MAP (OF.EQ . UNIT)
(OF . OFFICER))
```

```
(REQUIRED.CASES OF))
(DISTANCE (PATTERNS PAT.RELN.OF)
(RELATIONAL . T)
(TYPE . COMMON))
(PREDICATE.INDICATOR . DISTANCE)
(TD.MAP (TO . POS1)
(FROM . POS2)
(OF.EQ . DISTANCE))
(CLASS . LENGTHS)
(PATTERNS PAT.RELN.TO)
(PROPIT . T)
(REQUIRED.CASES TO FROM))
((PORT OF CALL) (TYPE . COMMON)
(RELATIONAL . SCALE)
(PATTERNS PAT.RELN.OF))
(CLASS . PORTS)
(PATTERNS PAT.RELN.OF)
(TD.MAP (OF . POS-OBJ)
(OF.EQ . POS)
(TIME . DATE))
(PREDICATE.INDICATOR . SHIP-PORT-ARRIVINGS)
(AUXILIARY.MAP (FOR . OF)))
```

Field names can also act like proper nouns, as in "the DFT field." This requires that two entries (for two word senses) be made in the lexicon.

Example:

```
(DFT    (TYPE . COMMON)
        (RELATIONAL . SCALE)
        (PATTERNS PAT.RELN.OF))
        (TD.MAP (OF.EQ . DRAFT)
                (OF . SHIP))
        (PREDICATE.INDICATOR . DRAFTINGS)
        (CLASS . LENGTHS))
(DFT 2 (TYPE . PROPERN)
        (CLASS . FIELDS))
        (ELEMENT . DFT))
```

2. Adjectives

Most adjectives* require the same attributes as relational nouns, but the value of the PATTERNS attribute is different. It may be any one of the following:

---

* Here we mean the grammatical category and not the modifiers in general. So, for example, nouns that can act as modifiers ("ballistic missile" in "ballistic missile sub") are added as nouns not adjectives

PAT.ADJ.OF -- used for adjectives that correspond to one-argument predicates; the adjective itself does not fill a case role. For example:

```
(NORMAL (NO.SELF . T)
(PREDICATE.INDICATOR . NORMALP)
(PATTERNS PAT.ADJ.OF)
(TD.MAP (OF . NORMATR)))
```

PAT.ADJ.OFEQ.OF -- the most frequently used pattern. The adjective fills the OF.EQ case. For example, the lexical entry for "American" which refers to the two-place predicate COUNTRY-OWNINGS in the the conceptual schema follows:

```
(AMERICAN   (CLASS . COUNTRIES)
            (ELEMENT . US)
            (PREDICATE.INDICATOR . COUNTRY-OWNINGS)
            (PATTERNS PAT.ADJ.OFEQ.OF)
            (TD.MAP (OF.EQ . OWNER)
                    (OF . OWNED)))
```

This indicates that the value of the OWNER case of COUNTRY-OWNINGS is indicated by the adjective (its value in this particular instance is given by the (ELEMENT . US) entry: US is the actual data base value) and the OWNED case is filled by the constituent that the adjective modifies.

PAT.ADJ.OFEQ.OF.TO -- for adjectives that correspond to three-place predicates. For example, "near" corresponds to a distance predicate that relates a distance (the OF.EQ case filled by "near" itself) and two positions. The OF case is filled by the thing (could be a physical object or a location) whose distance to the TO case is measured. The lexical entry for "near" follows:

```
(NEAR       (CLASS . LENGTHS)
            (SCALE . MINUS)
            (PREDICATE.INDICATOR . DISTANCE)
            (PATTERNS PAT.ADJ.OFEQ.OF.TO)
            (AUXILIARY.MAP (FROM . TO))
            (TD.MAP (OF.EQ . DISTANCE)
                    (OF . POS1)
                    (TO . POS2))
            (PROPIT . T)
            (REQUIRED.CASES OF TO))
```

The following are sometimes useful to avoid extraneous parses (but unlike above, not essential to getting a correct interpretation):

REQUIRED.CASES (list of obligatory cases), PROPIT (for words that allow an empty "it" as subject; e.g., "How far is it from Naples to Gibraltar.") AUXILIARY.MAP (pairs of surface prepositions and the thematic cases they take).

3. Verbs

Verbs, like relational nouns and adjectives, must have PREDICATE.INDICATOR, TD.MAP, and PATTERNS attributes. Several additional attributes are needed to handle the different kinds of syntactic patterns in which verbs occur and the corresponding mapping to conceptual cases. As a result, the entry of new verbs requires a great deal of linguistic knowledge.

DIROBJ -- T if can take a direct object. Similarly for INDIROBJ and indirect objects.

DIRECTION -- what surface case a directional pronoun can fill

PARTICLE -- for verbs that require a particle (e.g., "belong to," "head for").

INSEPARABLES -- like particle; used for verbs that do not allow particle to be separated from the verb (e.g.,"go into drydock").

NO.PASSIVE -- T for verbs that cannot be passivized).

AUXILIARY.MAP and REQUIRED.CASES -- same as described above.

B. Conceptual Schema and Linkage to The Data Base

The conceptual schema is used both to guide the semantic interpretation of utterances ("translation") and to provide an attachment to the data base so that data base queries can be constructed ("integration"). The information required for both functions is stored on the property list of the nodename for a schema node. For translation, properties are used to encode two basic kinds of links between nodes in the schema (actual names are given in uppercase in parentheses):

(1) subset/superset links (SUBSET, SUPERSET), and (2) predicate-argument links (ARG). In addition, there is a set of links used mainly for type-coercion (ID, OM, ATR).

Attachment to the data base is made through four properties of nodes in the schema: VR, KEY, VREST, VRMAP.

VR (virtual relation) specifies the data base relation to be accessed.

KEY is used only for nodes representing sets of objects ("domains" in data base terminology); it specifies a field (or fields) that uniquely identify members of the set. For example, UIC VCN for ships, LINEAL for officers.

VREST, a piece of SODA code that specifies a restriction on the virtual relation, is used to pick out a subset of entities in the relation. For nodes representing sets of objects, VREST provides the restriction that distinguishes entities in the relation that are members of the set from those that aren't (e.g., SUBMARINES are gotten from the SHIP relation with a restriction on the type field that specifies "SS" as the first two characters). For nodes representing predicates, VREST is used to identify those entities in the relation for which the predicate is relevant (e.g., a restriction on the type field is used to specify that only ships that aren't submarines can have a surface speed). VREST can of course be NIL.

VRMAP is used only for nodes representing predicates. It provides a mapping between predicate (case) arguments and data base fields. For each argument, VRMAP specifies the field (or fields) that contain the value of the argument for a particular entity.

For Example:

```
SUBMARINES: KEY (UIC VCN)
            SUPERSET (NAVAL-SHIPS)
            VR SHIP
            VREST (AND ((SHIP TYPE1) EQ 'S') ((SHIP TYPE2) EQ 'S'))

DRAFTINGS: ARG ((SHIP . SHIPS)            ;SHIP and DRAFT are the argument
                (DRAFT . LENGTHS))        ;names. SHIPS and LENGTHS are
           SUPERSET (PRED-HAVINGS)        ;nodes in the schema.
           VR SHIP
           VRMAP ((SHIP UIC VCN)          ;UIC VCN and DFT are DB fields
                  (DRAFT . DFT))

DOCTOR-CARRYINGS ARG ((CARRY-OBJ . NAVAL-SHIPS)
                      (CARGO . DOCTORS))
                 SUPERSET (CARRYINGS)
                 VR SHIP
                 VREST ((SHIP MED) NE '*')
```

```
VRMAP ((CARRY-OBJ UIC VCN)
       (CARGO . DOCTR))
```

Note that when a predicate is only applicable to a subset of entities in a relation, there are two ways of specifying the data base connection: (1) with a VREST on the predicate as in SURFACEP above, or (2) by constructing a schema node for the set satisfying the restriction (e.g., SUBMARINES, NAVAL-SHIPS) with the restriction on that node and specifying this new set as the argument to the predicate.

Several functions facilitate the creation of these properties. To save the results of this work do SAVE.DATA(), which will save the conceptual schema on a file named DATA.LSP needed for creating a new system. The function DN(node) deletes a node and all its links; the function EDITP can be used to edit the property list of the node.

The following discussion is done separately for objects and predicates.

1. Objects

   a. Creating Entries in the Conceptual Schema

      The function CREATE-OBJECT creates object entries.

      This function will prompt the user with a number of questions:

      Name: name for the node in the conceptual schema

      Subsets: names of nodes that represent subsets of this node

      Supersets: names of nodes that represent supersets of this node

      Is node a subset of THINGS? Answer should be yes for anything for which a "What" question can be asked (contrast with PERSONS for for which "who" questions get asked).

      Is node a subset of OBJECTS? Answer should be yes for any class which is a subset of physical objects (e.g., yes for SHIPS, no for COLORS).

b. Creating the Connection to the Data Base

The user should call the function CREATE-DBDOM and answer the following:

Domain: name of the data base node (same as response to Name for CREATE-OBJECT)

Virtual relation: name of the data base relation

Key roles: key fields in the relation that specify the object

Virtual relation restriction: a SODA fragment that specifies the restrictions on the relation for this object. The virtual relation name is used as a variable in the SODA expression.

Two different kinds of objects can be created; each requires a different set of responses for the corresponding call on CREATE-DBDOM.

* Case 1: the object specifies some subclass of a data base relation; the user must provide all the information above.
* Case 2: the object is only considered as an argument to some predicate, and in fact would typically be an argument to many predicates. For example, ASW (Antisubmarine warfare capability) could restrict a ship, an airplane, a weapon system. Then, one way to represent this information in the conceptual schema would be to create an object (ASW) which would be linked to the corresponding three predicates. In this case, the user only needs to specify some data base restriction. He then answers NIL to the virtual relation question, gives a dummy answer to the key role question, and uses that dummy answer in the SODA restriction.

Example of Case 1:

```
CREATE-OBJECT]
Name? OILERS
Subsets?
Supersets? NAVAL-SHIPS
CREATE-DBDOM]
Domain? OILERS
Virtual relation? SHIP
Key roles? (UIC VCN]
Virtual relation restriction? (AND ((SHIP TYPE1) EQ
    %'A')((SHIP TYPE2) EQ %'O']
```

Example of case 2:

```
CREATE-OBJECT]
Name? ASW
Subsets?
```

```
Supersets?  MISSILE-SYSTEMS
CREATE-DBDOM]
Domain?  ASW
Virtual relation?
Key roles?  MIS
Virtual relation restriction?  (OR (MIS EQ 'ASROC')(MIS EQ
     'SUBROC')
(MIS EQ 'MBU')(MIS EQ 'MK32'))
```

2. Predicates

To initially establish nodes for predicates, the user should use one of the following two functions:

CREATE-REL -- if there is no existing relation that this provides further restrictions on,

CREATE-SUBREL -- new relation is a specialization of some sort of an already existing predicate.

The function CREATE-DBMAP is also needed to establish the mapping between cases and field names.

For example:

```
_CREATE-REL]
Relation name?  FUEL-TYPE-RELS
FUEL-TYPE-RELS is not a DB node; do you wish to create it?
     Yes
Superset?
Subsets?
Argument (case/domain)?  FUEL FUEL-TYPES
FUEL-TYPES is not a DB node; do you wish to create it?  Yes
Argument (case/domain)?  SHIP SHIPS
Argument (case/domain)?
Case translation (old case/new case)?
Is FUEL-TYPE-RELS a sub-relation of HAVINGS?  ? No
CREATE-DBMAP]
Predicate?  FUEL-TYPE-RELS
Virtual relation (relation name, IGNORE, SUBFORM, or FN)?
     SHIP
Relation restriction?
Case to role mapping:
Role name for SHIP?  UIC VCN
Role name for FUEL?  FTP2
```

The following functions must also be used:

CREATE-ATR: it links entities to their attributes. For example, given a MISSILE-TYPINGS predicate that links

MISSILE-TYPES and MISSILE-SYSTEMS, CREATE-ATR must be used to specify that MISSILE-TYPES is an attribute of MISSILE-SYSTEMS.

CREATE-ID -- for predicates that are subsets of NAMINGS, such as CONVOY-NAMINGS.

CREATE-OM -- for predicates that establish owner-member relationships such as CONVOY-MEMBER.

VI D-LADDER ERROR MESSAGES AND ERROR CONDITIONS

A. Errors in Interpreting the Inputs

Four stages are involved in the interpretation of inputs by D-LADDER: (1) parsing, (2) translation, (3) integration, and (4) data base query construction. The system prints a message to the user as it passes through each stage so that the point at which an error is incurred may be identified.

Parsing will succeed as long as each word in the input is in the D-LADDER lexicon and a correct syntactic structure can be built using the information in the lexical entries for the words. D-LADDER prints an ^ under each word as it parses. It will indicate unknown words at this point with the message:

```
The input can't be interpreted.
[Word] is an unknown word.
The longest phrase constructed was: [portion of sentence]
Please try again.
```

The message "translating..." is printed when D-LADDER starts to build a semantic interpretation for the utterance. It can fail to do so if there is incorrect or missing information in the conceptual schema for the concepts referred to by the words in the input. If this happens D-LADDER will print out

```
translating...failed.

The input can't be interpreted.
The longest phrase constructed was:  [input sentence]
Please try again.
```

The two places to look for correcting such problems are in the lexical entries for the words, especially at the semantic attributes PREDICATE.INDICATOR and TD.MAP, and at the conceptual schema nodes these attributes access.

The integration stage mainly involves handling noun-phrase references, and it is unlikely the system will fail. If it does, the following messages will be printed:

```
translating...integrating...failed.
The input can't be interpreted.
The longest phrase constructed was:  [input sentence]
Please try again.
```

The best at this point is for the user to try a rephrasing and report the error to the D-LADDER implementers.

When the integration stage is completed, the message,

"integrating...succeeded!"

followed by information about the parsing time will be printed.

If a successful data base query can be constructed, the system will print out a paraphrase of the input and then connect to the Datacomputer. Otherwise, it will respond with, "I can't interpret this query." Problems at this level are a result of some part of the data base attachment (i.e., the VR, VREST, VRMAP properties) not being correct.

B. Errors in Querying the Data Base

Many things beyond the control of D-LADDER or the user can go wrong in accessing data from a remote computer over the ARPANET. D-LADDER attempts to recover from many of these error conditions automatically, and will inform the user as it does so.

When D-LADDER establishes a network connection to an instantiation of the Datacomputer data base management system, it prints out "CONNECTING TO DATACOMPUTER AT <site>," where <site> is the remote computer. If the remote site is operational but the Datacomputer there is not, an error condition will be noted rather quickly. If, however, the remote site is not operational at all, D-LADDER will not be aware of a problem until the attempt to establish a network connection times out. (The time required for this to occur is a set parameter of the local operating system under which D-LADDER is running.)

In any case, when a site is found to be inaccessible, D-LADDER will attempt to access a backup site, with appropriate notification to the user. When its set of backup sites has been exhausted, an appropriate message is printed.

In a similar manner, if an error condition ensues when an attempt is made to open a file, D-LADDER will attempt to access a backup file (which may involve establishing a new network connection to another site). Again the user is kept informed of what is happening. If no backup is available for a given file, a message to that effect is written and the query is aborted.

C. D-LADDER Bugs

Should D-LADDER simply break because of a program bug, the system will most likely print out some obscure message, as well as the next event number followed by a semicolon instead of a left arrow. In this case, type control-D to restore the left arrow and then type RESET.

It is possible but unlikely that D-LADDER will simply "hang" in the middle of processing some request. This is often due to an extraordinary load on either the machine running D-LADDER or the machine running the Datacomputer software. Should you lose patience, typing control-D and then RESET, as indicated above, will restore D-LADDER to a state in which it can process new inputs. Care should be taken not to do this too often, because it tends to make D-LADDER run out of memory space.

Appendix A

DEFINING NEW GRAMMAR AND LEXICON

Appendix A

DEFINING NEW GRAMMAR AND LEXICON

DIAMOND is a system for defining how natural-language inputs are to be interpreted. Rules for interpreting inputs are written as phrase-structure rules augmented with procedures to be executed at successive stages during processing. The result of an interpretation is one or more 'parse trees,' which reflect the phrase structure of the input and which have associated with them attributes of the phrases (as specified by the procedure).

1. Defining Rules

a. Form of the Rules

DIAMOND rules consist of a phrase-structure rule and properties that can be procedures to be evaluated during processing. There are currently three processing phases and the procedures for those phases are CONSTRUCTORs, TRANSLATORs, and INTEGRATORs. CONSTRUCTOR procedures are evaluated during bottom-up assembly of phrases. TRANSLATOR procedures are evaluated after a complete parse tree has been built, and INTEGRATOR procedures are evaluated after the TRANSLATORs.

TRANSLATORs are evaluated starting with the TRANSLATOR for the root phrase. In normal processing, the TRANSLATORs of each constituent are evaluated before that of the root phrase. This can be overridden by setting the global variable DO.NOT.TRANSLATEALL to T, and then explicitly evaluating TRANSLATORs for individual constituents with the functions TRANSLATE(phrase) or TRANSLATEALL(). If a rule does not have a TRANSLATOR associated with it, normally none of its constituents will be translated Setting TRANSLATE.ANYWAY to T overrides this and in that case all TRANSLATORs are evaluated. Normally, the results of the evaluation of a TRANSLATOR for a phrase are kept with that phrase and

are available to all 'parent' phrases (if it appears in more than one interpretation). This sharing of translations among multiple interpretations can be overridden for all phrases by setting the global flag RETRANSLATEALL to T, or it can be overridden for individual phrases by setting the property RETRANSLATE on individual rules to T.

INTEGRATORs are evaluated starting with the INTEGRATOR for the root phrase. Normally, the INTEGRATORs for each constituent are evaluated before that of the root phrase. This can be overridden by setting the global variable DO.NOT.INTEGRATEALL to T, and then evaluating INTEGRATORs for individual constituents with the functions INTEGRATE(phrase) or INTEGRATEALL(). If a rule does not have an INTEGRATOR associated with it, the default is not to integrate any of its constituents. Setting INTEGRATE.ANYWAY to T overrides this and causes all INTEGRATORs to be evaluated. Generally, the results of evaluating the INTEGRATOR for a phrase are kept with that phrase and are available to all its 'parent' phrases. This sharing of 'integration' results among multiple interpretations can be overridden for all phrases by setting the global flag REINTEGRATEALL to T, or it can be overridden for individual phrases by setting the property REINTEGRATE on individual rules to T.

CONSTRUCTORs, TRANSLATORs, and INTEGRATORs are stored as properties of the rule. If they are a list they are EVAL'd, otherwise they are treated as a function to be called.

A rule can also have a COMMENT property, which is not used in processing, but is provided for documentation purposes. Other properties can be given to a rule for descriptive purposes, and they can be saved if the property name is added to the list RULEPROPS.

b. Attributes and Factors

i. Attributes

(1). Referencing Attributes

Phrase attributes are accessed by a special function "@" which takes any number of arguments. The arguments of @ are the attribute(s) desired. Phrase constituents are stored as attributes of the phrase, so phrase consitutents are also referenced with @. For example:

(@ NUM) returns the value of the NUM attribute

(@ NUM NP) returns the NUM of the constituent named NP

(@ NP) returns the constituent named NP

(@ NUM ! X) returns the NUM of the phrase stored in variable X. Thus after (SETQ X (@ NP)), this is equivalent to (@ NUM NP)

(@ : X NP) returns the value of the attribute whose name is stored in variable X. Thus after (SETQ X (QUOTE NUM)) it iis the same as (@ NUM NP).

The general form of an '@' statement is (@ name . tail); name specifies the attribute name, tail specifies the phrase. The currently active phrase is stored in the variable named SELF. Thus (@ NUM) is equivalent to (@ NUM ! SELF), and in general if any list doesn't end with ! X, then ! SELF is automatically added. All arguments before ! are taken literally unless they follow a ":."

(2). Setting Attributes

Attributes are set with the function @SET. For example:

(@SET COUNT (ADD1 (@ COUNT))) increments the COUNT attribute

(@SET NUM (QUOTE PL) NP) sets NUM to PL in constituent NP

The general format is (@SET name value . tail). Where 'name' is taken literally unless preceded by "$," value is always evaluated, and tail is treated like tail of the @ argument list--in which ! can be used.

Attributes can be copied up from constituents with the function @FROM. Thus, (@FROM C ATTR1 ... ATTRn) is equivalent to

```
(PROGN (@SET ATTR1 (@ ATTR1 C))
       .
       .
       .
       (@SET ATTRn (@ ATTRn C)))
```

ii. Factors

Factors are set with the function @FACTOR. The general form is (@FACTOR name value). Name and value are treated like they are in @SET, except name does not become a separate attribute of the phrase but is added as a factor. Specifically, NAME and VALUE (for non NIL values) are added to the front of the phrase attribute lists FACTORNAMES and FACTORVALUES, respectively. Inside a CONSTRUCTOR, the factors are set on SELF. Inside a TRANSLATOR or INTEGRATOR, they are set on a variable called ROOT, which is equivalent to SELF except that it has translation-time factors on it.

The function F.REJECT(factor) is used to reject a phrase. F.REJECT can be broken during debugging using BREAK0(F.REJECT (EQ FACTOR factorname)) to look for a certain factor.

iii. System-Defined Attributes

Several attributes of phrases are defined by the system. They include:

CATEGORY--The category record. For example, (@ DIAMOND.NAME CATEGORY ! X) returns the category name of the phrase named by the value of the variable X.

DIAMOND.RULE--The rule record. For example, (@ DIAMOND.NAME RULE ! X) returns the rule name for X.

DIAMOND.LEFT--The leftmost position of a phrase (any integer).

DIAMOND.RIGHT--The rightmost position of a phrase.

DIAMOND.SONS--The list of constituent phrases.

DIAMOND.FATHER--The immediately dominating phrase. It is NIL for a root phrase or when it is not used inside a TRANSLATOR or INTEGRATOR

DIAMOND.SPELLING--The spelling of the word if it is a terminal phrase.

SCORE--the score of the phrase if any factors are specified.

c. BNF of Rule

```
   RHS      = RHSALTS
   RHSALTS  = RHSSERIES ("/" RHSALTS)
RHSSERIES   = RHSITEM (RHSSERIES)
   RHSITEM  = "(" RHSSERIES ")"  /                    option
              "{" RHSALTS "}"  /                    alternatives
              STRING ("#" name)    /                  literal
              ID ({"#" name)  /
                     "=" { "{" category "/" category ... "}" /
                              category})
```

Examples:

NP#1 for noun phrase
PRED = {VP / ADJP} PRED is name for each alternative

or

S = C = {NP#1 / PP} MODAL SUBJECT = NP#2 (HAVE)(BE "ING") VP ;

2. Functions for Defining and Editing Rules

a. Functions

RULES.DEF is like the INTERLISP function DEFINE. It takes a list of rule definitions each of which is of the form

(rulename rulecategory . tail)

The initial segment of TAIL up to a semicolon is used as RHS of the rule. An = at the front of the tail is presently discarded. Items on TAIL after the semicolon are used to set properties of the rule--each pair is used as NAME & VALUE. If the rule is already defined, this redefines it.

RULES.DEFQ is like the INTERLISP function DEFINEQ.

RULEPROC.DEF(rulename property value) defines or redefines a property on a particular rule (especially CONSTRUCTOR, TRANSLATOR, and INTEGRATOR).

RULEPROC.DEFQ is NLAMBDA version.

EDITRHS(rulename) allows editing of the right-hand side of the rule (i.e. the phrase structure part).

EDITRULEPROP(rulename property) is for editing a particular property of a rule.

DELETE.RULE(rulename) deletes a given rule.

SHOWRULE(rulename) prints out a particular rule.

SHOWCATEGORYRULES(categoryname) prints all rules in the specified category.

RULESUSING(category) prints the names of all rules in which the category is used.

CHANGERULENAME(from to) changes the name of the rule indicated.

SHOWPHRASES(file comments) will print out all phrase structure rules. If comments = T, then the COMMENT property on the rule will also be printed. If the file is not specified, the rules will be printed on the terminal.

b. Examples

```
_RULES.DEFQ((NP1 NP ADJ N ; CONSTRUCTOR (attribute and factor stms)
   TRANSLATOR (TRANSLATOR code)))
(NIL)
_SHOWRULE(NP1)

(NP1      NP = ADJ N ;
      CONSTRUCTOR (ATTRIBUTE AND FACTOR STMS)
      TRANSLATOR (TRANSLATOR CODE))
NP1
_RULEPROC.DEF(NP1 COMMENT (THIS IS A COMMENT]
(THIS IS A COMMENT)
_SHOWRULE(NP1)

(NP1      NP = ADJ N ;
      CONSTRUCTOR (ATTRIBUTE AND FACTOR STMS)
      TRANSLATOR (TRANSLATOR CODE)
      COMMENT (THIS IS A COMMENT))
NP1
```

```
_EDITRHS(NP1)
edit
P
(ADJ N)
*
(-1 DET)
*P
(DET ADJ N)
*(N PREPP)
*P
(DET ADJ N PREPP)
*OK
NP1
_EDITRULEPROP(NP1 CONSTRUCTOR)
edit
*P
(ATTRIBUTE AND FACTOR STMS)
*(4 STATEMENTS)
OK
CONSTRUCTOR
_RULES.DEFQ((NP2 NP "THE" N ; ]
(NIL)
_SHOWCATEGORYRULES(NP)

(Rules for NP

(NP1      NP = DET ADJ N PREPP ;
      CONSTRUCTOR (ATTRIBUTE AND FACTOR STATEMENTS)
      TRANSLATOR (TRANSLATOR CODE)
      COMMENT (THIS IS A COMMENT))

(NP2      NP = "THE" N ;)

)
NIL
_RULESUSING(DET)
(NP1)
```

3. Defining and Editing Words and Categories

a. Functions for Categories

DELETE.CATEGORY(category) deletes all declarations for an entire category.

EDITCATEGORYPROP(category prop) allows editing of a property for a given category.

WORDPROCS.DEF(category attribute expression) sets properties and expressions to be evaluated like RULEPROC.DEF but for word categories.

WORDPROCS.DEFQ nlambda version of WORDPROCS.DEF

CHANGECATNAME(from to) allows changing the name of an entire category.

b. Word Definitions

Words are defined as part of the definition of their category. Procedures can be declared that are to be applied to all words in that category.

i. Functions

WORDS.DEF((category . tail)) adds new words to the category. Each item in TAIL is a list of the form (lexitem . lextail). LEXITEM is either an atom or a list of atoms (for multiword lexical entries). For multiword lexical entries, all the words in the list are treated as a single lexical item with blanks between words. LEXTAIL can start with an integer(sense number) to disambiguate different senses of the LEXITEM. Other items on LEXTAIL are (name . value) pairs for attributes.

DELETE.WORD.DEF(word category sensenumber) deletes a word definition. If CATEGORY is NIL, then all definitions for word in all categories is deleted.

EDITWORD(word category sensenumber) edits a word definition

ADDWORDATTR(category word attribute value) adds the attribute with the indicated value to the word entry in indicated category.

ADDMULTATTRS(category wordlist attr value) does ADDWORDATTR for each word in wordlist. Note that the same value is given for each occurrence of the attribute.

SHOWCATEGORYWORDS(category) displays all the words in a category.

SHOWWORD(word) will show the entry for an individual word.

SHOWWORDS((word1 word2 ..)) will show the entries for all words in the list.

c. Morphology

i. Declaring Suffixes

SUFFIXLIST has the suffixes in it and how they appear in the grammar, as well as the categories they can appear with. It has the form ((suffix grammarform cat cat) ....) e.g.,

((ES S N V)(ED ED V)(TEEN TEEN DIGIT))

INVERTSUFFIX is list of suffixes to invert e.g., (S ED EN)

DROPSUFFIX is list of suffixes to drop

ii. Irregular Forms

Defining irregular forms:

(IRREGULARS.DEFQ category (irregular (root suffix))(irreg (root suffix)))

or

IRREGULARS.DEF((cat (irregular (root suffix))))

e.g.,

```
(IRREGULARS.DEFQ V (WENT (GO ED))
                   (GONE (GO EN))
                   (RAN (RUN ED)))
```

d. Example

For example:

```
_WORDS.DEF((N (BOLT (SEMANTICS . SBOLT))
               ((ALLEN WRENCH) (SEMANTICS . A.WRENCH]
N
_IRREGULARS.DEF((N (CHILDREN (CHILD S))))

_SHOWCATEGORYWORDS(N)

(Words for N
    ((ALLEN WRENCH) (SEMANTICS . A.WRENCH))
    (BELT)
    (BOLT       (SEMANTICS . SBOLT))
    (BOX)
    (PLATFORM)
    (PULLEY)
    (PUMP)
```

```
    (SCREW)
    (WRENCH))

(Irregular forms in N
    (CHILDREN (CHILD S)))

NIL
```

4. Manipulating Files

Once a 'language definition' has been specified, the information can be saved in one or more files (which can be later reloaded).

a. Saving On a Single File

To save everything on one file use SAVEGRAMMAR(filename). If no file is specified, then the file is written as GRAMMAR.LSP.

b. Separate Files

A language definition can be separated into several files that are referenced through an index. An index consists of a list of filenames. Associated with each filename is a variable filenameCATS (e.g., NP.RULESCATS) which specifies the contents of that file. The variable consists of a list of the categories to put in the file. If only the category name appears, then all the category information--rule, word, and category declarations--are written on the file.

Alternatively, to separate words from rules, a category can be specified along with the information (rules or words) to be saved for that category. For example, if (Np.RULESCATS) is set to

```
((RULES NP)NOMHD NOM (RULES N))
```

then all of NOMHD and NOM will be saved on NP.RULES along with the rules for NP and N. If NP.WORDSCATS is set to

```
((WORDS NP)(WORDS N))
```

then the word declarations for NP and N will be saved on the file NP.WORDS.

The function SAVEINDEXED(file compileflag) will save an entire set of files specified by an index as well as create an index file, and will optionally compile the file if compileflag is set to T (see below).

SAVECATEGORIES(file compileflag) will save a list of categories specified by fileCATS on the file 'file' and optionally compile it.

c. Incremental Changes

SAVECHANGES(flag) will produce new files for all files containing categories that have been altered. If flag is T, then the file will also be compiled.

d. Loading Files

LOAD(file) loads a grammar saved with any of the above functions. LOAD of an index file will automatically load all files in the index.

e. Compiling Rules

COMPILE.RULES((RULELIST) FILENAME) compiles the CONSTRUCTORs, TRANSLATORs and INTEGRATORs for all the rules on the list and puts the compiled function call in place of the expression under the appropriate property. The expression is saved under the property CONSTRUCTOR.EXPR (or TRANSLATOR.EXPR or INTEGRATOR.EXPR). If a file name is given the functions are put on the file, but the file IS NOT CLOSED.

COMPILE.RULEFILE(filename) compiles the CONSTRUCTORs and TRANSLATORs for all the rules on the file given. If that file has been loaded, the CONSTRUCTORs are taken from core, otherwise the rules are loaded. The compiled definition is put in place of the 'expr' on the rules. The compiled file is filename.COM-extension. For example, NP.RULES will have a compiled file NP.COM-RULES.

COMPILE.GRAMMAR(index) compiles all the rule files in either the list INDEX or in the global GRAMMARINDEX (which comes from a grammar index file).

SAVEINDEXED, SAVECATEGORIES, SAVECHANGES all take an extra argument, which if T, will compile the CONSTRUCTORs for the rules being written, and mark the new files with the name of the compiled file, so the compiled functions will be automatically loaded. This information appears as (Comfile filename) after the date in the file. If there is an expression stored under the property CONSTRUCTOR.EXPR, that property will be saved as CONSTRUCTOR.

When a file is loaded, if it contains a pointer to a compiled file, the compiled file will be loaded and the CONSTRUCTOR, TRANSLATOR, and INTEGRATOR replaced with the compiled version.

If the global variable DESTROY.EXPRS is NIL, then the exprs will be saved under the xx.EXPR property, otherwise they will be deleted. The setting is usually NIL for debugging, but setting it to T frees space for 'production' systems.

LOADCOMRULES(filename) will load a compiled file and replace the expressions under the CONSTRUCTOR, etc. properties with calls to the compiled functions.

If a compiled CONSTRUCTOR is edited, EDITRULEPROP will retrieve the the expression and delete the call to the compiled function. It will NOT automatically recompile the expression as a function.

5. Parsing Sentences

There are three top-level functions for the system: TP, DP, and DIALOG. The function TP will parse an individual sentence and produce an interpretation. The function DP will interpret a sentence and then call the function DIALOG.RESPONSEFN with the result. In the NSFSYS sysout, this function interacts with the task model and produces a reply. The function DIALOG is a read-loop of calls on DP.

For example:

```
_TP((THE ALLEN WRENCH IS IN THE BOX]
45 S1
  9 NP2
    2 "THE"
```

```
    8 NOM1
      7 NOMHD1
        6 N ALLEN WRENCH
  11 BE1
    10 BE IS
  43 PREDP1
    42 PP1
      12 PREP IN
      41 NP2
        22 "THE"
        40 NOM1
          39 NOMHD1
            38 N BOX
1 INTERPRETATIONS 50 PHRASES
415
```

Note the use of the multiword "allen wrench" here.

To look at attributes of the completed phrases, use SHOWATTRS(phrasenumber (optional list of attrs)). If no attributes are given, then the global list DISPATTRS is used. This should be set to the attributes desired to be viewed. For example:

```
_SHOWATTRS(7)

COMPOUND    =    NO
```

To look at factors use SHOWFACTORS(phrasenumber).

SHOWALLINTERPS() will print all the interpretations of an input.

SET(SHOWPARSETREE) stops printing of the parse tree when a parse is completed. SET(SHOWPARSETREE T) restores printing.

a. Debugging

To trace a parse do SET(TRACEMODE n) where n=1 gives a 'short' trace, n=2 giives a longer trace. For n>1, the attributes of a phrase on the list DISPATTRS will also be displayed.

To turn off tracing, do SET(TRACEMODE 0)

To break F.REJECT, either use BREAK(F.REJECT) to break it every time or BREAK0(F.REJECT (EQ FACTOR nameoffactor)).

Once inside the 'break' PNS() will print out the current phrase being built, and its constituents.

SHOWATTRS() will show all the attributes of the phrase whose names are on the list DISPATTRS, OR

(SHOWATTRS SELF '(attribute attribute)) will show the attributes in the list.

FACTOR (a variable) will print the name of the factor for which the F.REJECT was called.

(@ NAME RULE) will print the name of the rule the F.REJECT is in.

To continue type OK or GO

6. Fitting It All In

DESTROY.EXPRS

No spelling tree/Jonathan's stripper

7. Paraphrasing

Paraphrases are input using TP or the function PARAPHRASE.DEF(new old) . For example, a paraphrase for "in the box" can be created by:

```
_TP((LET ITB MEAN IN THE BOX]
```

(note here that LET and MEAN are required words)

```
Delete prior definition of ITB? Yes or No : Yes
PARAPHRASE#1

or PARAPHRASE.DEF(ITB (IN THE BOX))

_SHOWRULE(PARAPHRASE#1)

(PARAPHRASE#1
          PP = "ITB" ;
      CONSTRUCTOR [PARAPHRASECONSTRUCTOR
                   (QUOTE (PP (PREP IN)
                              (NP (LITERAL.THE THE)
                                  (NOM (NOMHD (N BOX]
      TRANSLATOR PARAPHRASETRANSLATOR
      PARAPHRASEDEF ((ITB)
                     (IN THE BOX)))
PARAPHRASE#1
_TP((THE PUMP IS ITB]
27 S1
```

```
  9 NP2
    2 "THE"
    8 NOM1
      7 NOMHD1
        6 N PUMP
  11 BE1
    10 BE IS
  25 PREDP1
    17 PARAPHRASE#1
      16 "ITB"
256
```

The rule constructed for it can be seen by looking at the rule named PARAPHRASE#n.

Paraphrases can only be constructed for full constituents. Thus a paraphrase can be constructed for 'the box' or 'in the box' but not 'is in the box' or 'the bolt is'.

a. Ellipsis

Ellipsis is done within the parsing system. The elliptical expression is substituted for the largest possible constituent in the preceding input. For example,

```
_TP((THE PLATE))
41 S1
  9 NP2
    2 "THE"
    8 NOM1
      7 NOMHD1
        6 N PLATE
  21 BE1
    20 BE IS
  36 PREDP1
    28 PARAPHRASE#1
      27 "ITB"
441
```

8. Splitting Phrases

(SPLITPHRASE n) where n is an integer allows splitting into several alternatives. If n is 0, it calls F.REJECT; if n is 1, it acts as a no-op and returns 1, if n is greater than 1, it causes control to split n ways and returns an integer between 1 and n which is different in each incarnation.

(SPLIT.CHOICE list) uses SPLITPHRASE to return one of the elements from list or F.REJECT if list is NIL. For example, (@SET NBR (SPLIT.PHRASE '(SG PL))) will continue from this point with two alternatives, one with NBR set to SG, and the other with NBR set to PL.

9. Miscellaneous

To initialize the world, call INITIALIZEDIAMOND().

To set the root category for the grammar, call SETROOTCATEGORY(categoryname); or if there are several root categories, call SETROOCTCATEGORIES(list of cats).

REFERENCES

1. D. E. Walker (ed.) Understanding Spoken Language. Elsevier North-Holland, NY (1978).

2. A. E. Robinson, D. E. Appelt, B. J. Grosz, G. G. Hendrix, and J. J. Robinson, "Interpreting Natural-Language Utterances in Dialogs about Tasks," Technical Note 210, Artificial Intelligence Center, SRI International, Menlo Park, CA (March 1980).

3. G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz and J. Slocum, "Developing a Natural-Language Interface to Complex Data," ACM Transactions on Database Systems, Vol. 3, No. 2 (June 1978).

4. D. Sagalowicz, "IDA: An Intelligent Data Access Program," Proc. Third International Conference on Very Large Data Bases, Tokyo, Japan (October 1977).

5. P. Morris and D. Sagalowicz, "Managing Network Access to a Distributed Data Base," Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California (May 1977).

6. W. Teitelman, "INTERLISP Reference Manual," Xerox Palo Alto Research Center, Palo Alto, California (December 1975).

7. W. H. Paxton, "A Framework for Speech Understanding," Technical Note 142, Artif, Menlo Park, CA (June 1977).

8. J. J. Robinson, "DIAGRAM: a Grammar for Dialogues," Techcal Not 205, Artificial Intelligence Center, SRI International, Menlo Park, CA (February 1980).

9. Naval Electronics Laboratory Center, "The Relational Model for the Blue File Data Base (Revised)," Project Scientist: Garrison Brown, S Diego,lifornia (November 1976).

10. Computer Corporation of America, "Datacomputer Version 1 User Manual," Cambridge, Massachussetts (August 1975).