A TRANSPORTABLE NATURAL LANGUAGE FRONT-END TO DATA BASE MANAGEMENT SYSTEMS

by

Steve J. Safigan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

Approved:

Dr. J. Terry Nutter, Chairman

Dr. John W. Roach

Dr. Sallie Henry

September, 1987

Blacksburg, VA

A TRANSPORTABLE NATURAL LANGUAGE FRONT-END TO DATA BASE MANAGEMENT SYSTEMS

by

Steve J. Safigan

Dr. J. Terry Nutter, Committee Chairman
Department of Computer Science

(ABSTRACT)

Although some success has been achieved in the design of front-end natural language processors to data base management systems, transporting the processor to various data base management systems has proven to be elusive. A transportable system must be modular; it must be able to adapt to radically different data domains; and it must be able to communicate with many different data managers. The system developed accomplishes this by maintaining its own knowledge base distinct from the target data base management system, so that no communication is needed between the natural language processor and the data manager during the parse. The knowledge base is developed by interviewing the system administrator about the structure and meaning of the elements in the target data base. The natural language processor then converts the natural language query into an unambiguous intermediate-language query, which is easily converted to the target query language using simple syntactic methods.

ACKNOWLEDGEMENT

I would like to thank Dr. J. Terry Nutter, my thesis advisor, for her extensive guidance, direction, and review of the thesis material. I'd also like to thank the other committee members-- Dr. John W. Roach and Dr. Sallie M. Henry-- for their review and comments. Finally, I'd like to thank my superiors and the support staff at the Naval Surface Weapons Center for allowing me to participate in the advanced training program.

TABLE OF CONTENTS

1. INTRODUCTION

Conventional data base management system (DBMS's) have proven useful for storing information and handling user queries and updates. Such systems allow the computer to accept and store information in a structured format, and retrieve the information when requested. With all but the most advanced DBMS's however, the user must add, modify, and retrieve information using a formal data base language. Such languages are unambiguous and precise, forcing the user to query the data base in a fashion to which he is not accustomed.

Natural language interfaces to data bases allow users to communicate with a data base manager in the user's most natural and effective means of communication. Such systems allow the user to formulate a query in his own words, placing the burden of interpretation on the computer. Although some success has been achieved in the development of such systems, they generally suffer from two distinct limitations.

First, most natural language interfaces are developed for specific domains of data, and cannot be easily transported to data bases that deal with different data

domains. Therefore, a great deal of effort must be expended in order to customize a natural language interface to accommodate a new data base. This duplication of effort makes it unreasonably difficult to customize the natural language interface to target data domains, even if the same data base management system is used to manage all data bases.

Second, most natural language understanders are integrated into a custom data manager, allowing free communication between the natural language processor and the data manager, but making it very difficult to transport the natural language system to a new data manager. This means that a natural language front-end must be customized for each individual data base management system. This is a costly solution, since the effort required to design and develop a natural language processor can greatly exceed the effort required to design and develop the data base management system itself.

This thesis addresses these problems by designing a natural language interface which is both domain- and data-manager independent. This poses many technical and theoretical problems, given the complexity and ambiguity of natural language. Researchers are just beginning to

address these problems. Hence, no approach to transportable natural language interfaces has received a wide degree of acceptance.

In the work reported here, transportability is accomplished by developing an architecture which modularizes the natural language processor and restricts communication between the natural language processor and the data manager to well-defined, easily customizable communication channels. Since this style of transportable natural language processor has limited access to the data manager and no access at all to the actual data, it must maintain its own knowledge base. Such a knowledge base must contain information about the structure and meaning of the target data bases. The natural language processor must use this knowledge base in order to reason intelligently about the meaning of a query, and transform the query into a formal data base query. All ambiguity must be resolved, and the formal query must be organized in the same way as the data is structured in the target data base(s).

The knowledge base must be small compared to the overall size of the data base. This is important, since the additional space and effort required to establish and

maintain a substantial knowledge base can easily approach the space and effort required to store and maintain the data bases themselves. Because it must be transportable, the knowledge base must also be able to accept and store information about new data domains. This is very difficult, because a substantial amount of background information is needed in order to resolve context-related ambiguity.

This thesis examines the theory and methodology of an architecture for a transportable front-end natural language processor and discusses its capabilities and limitations. Chapter 2 discusses the various architectures used with existing systems, and how these architectures help or inhibit their transportability. Chapter 3 introduces the proposed architecture. Chapter 4 discusses the information requirements of the natural language processor's own knowledge base. Chapter 5 introduces the interface expert, which gathers the information needed by the knowledge base in order to recognize a new data domain.

Since the natural language processor must be transportable across data base managers, it does not know the syntax of the targeted data base query language in advance. Therefore, an intermediate query language is

developed which can be transformed into most common data base query languages using simple syntactic methods. The requirements for the intermediate language are defined in chapter 6. The parser used to parse the sample natural language queries into the intermediate query language is defined in chapter 7. Proposed extensions are outlined in chapter 8.

2. BACKGROUND

Natural language interfaces to data bases arose out of early research in automatic language translation. The first natural language understanders used their own internal knowledge base to store data, instead of using an external data base management system. When data base management systems became widespread in the 1960's, interest grew in providing a natural language interface to these systems. But the problems of natural language understanding were difficult, and the theory required to construct useful natural language interfaces to data bases did not exist.

In the 1970's, progress in interpreting natural language made it possible to develop reasonable natural language interfaces. Interfaces such as SHRDLU [Winograd72] and LADDER [Hendrix77] successfully interpreted a wide range of syntactic structures (see section 2.1 below). But researchers found it particularly difficult to transport these interfaces to other data domains without sacrificing much of the understanding. This is because much of the background information is domain-specific, and is not applicable to other data domains. Therefore, natural language interfaces were limited to a single data manager with very specific, well-defined data domains.

Today, researchers are beginning to develop theories of making natural language interfaces transportable to different data domains and even to different data base management systems. Systems such as TEAM [Grosz87], IRUS [Bates83], ASK [Thompson85], and Ginsparg's system [Ginsparg83] have shown limited transportability. (See section 2.3 below.)

The theory of data base management has advanced over this period as well. Deductive data base management systems have been developed. These systems employ sophisticated knowledge representation techniques in order to manipulate data as knowledge, rather than as information. Most use an associative network style of linking one element of information to another, rather than the more conventional file-field method of relating information. Such knowledge-based systems will greatly aid in the development of natural language systems, because the architecture of an associative network more closely matches the structures which can be produced by parsing natural language. Also, background information can be added to an associative network structure. However, such systems are still a topic of research, and not yet in widespread use. We will therefore ignore the capabilities of deductive data bases, and concentrate

instead on the problems and strategies of developing an interface to the more standard data base management systems in use today.

2.1. NON-TRANSPORTABLE NATURAL LANGUAGE INTERFACES

The earliest natural language interfaces were applied to a single domain, such as the "blocks world" of SHRDLU [Winograd72]. In SHRDLU, the natural language interface understands queries and commands dealing with a robot arm and a set of blocks of different shapes, colors, sizes, and positions in relation to each other and to the table top. SHRDLU can "remember" where each block is placed, and understand and act upon complex commands such as, "Place the large green block on the square block." It can also handle anaphora resolution, such as "Now place it on the table." SHRDLU has no need to interface to a formal data manager, since the small amount of information about blocks world can fit into SHRDLU's own knowledge base. The architecture for SHRDLU is outlined in Figure 1.

The parser accepts the English-language query and parses the query into first-order logic with the help of background knowledge and facts in the knowledge base. If

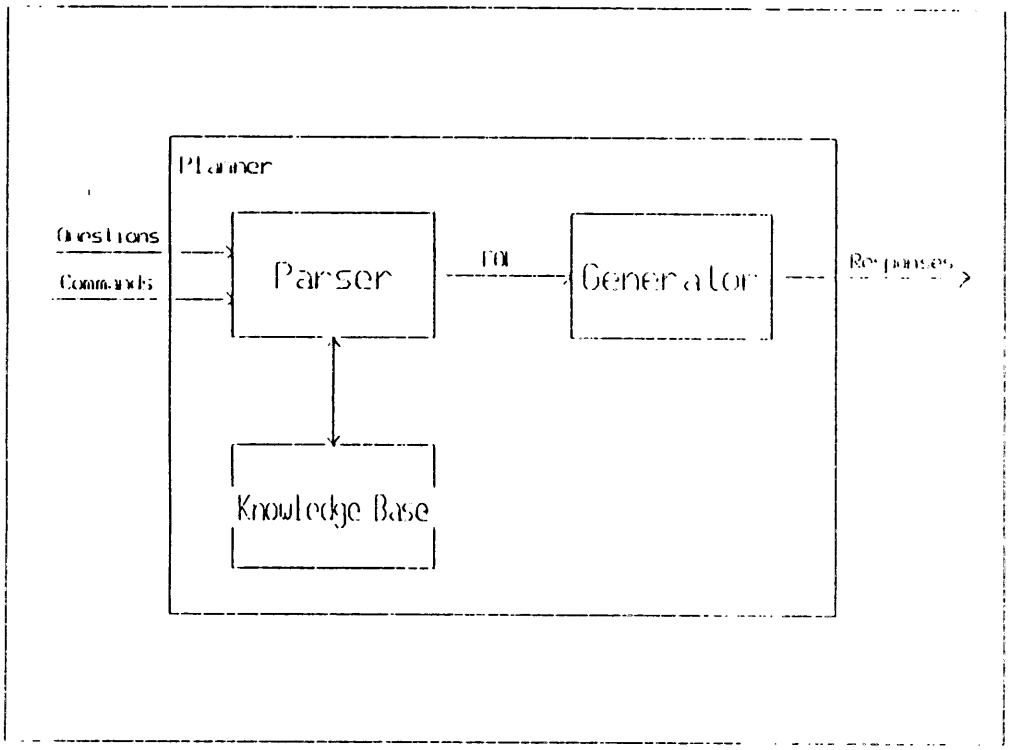


Figure 1: Architecture of SHRDLU

the result of the parse dictates a change in the knowledge state, then that change is noted in the knowledge base. Finally, a response is generated in first-order logic and passed to the generator, which generates an English-language response.

Despite its success within the domain of blocks world, SHRDLU is not easily transportable to different data domains which have no relevance to blocks world. Its static background knowledge and method of knowledge representation make it very difficult to adapt the SHRDLU system to other domains. Other systems developed at the time used similar domain-dependent architectures. One such system is LADDER [Hendrix77,Hendrix81b], which deals with the domain of ship information and other selected topics. Another is LUNAR [Woods78], which deals with the chemical analyses of the Apollo 11 moon rocks.

2.2. NATURAL LANGUAGE INTERFACES TRANSPORTABLE ACROSS DATA DOMAINS

Several natural language systems which are portable across data domains have been developed. A prototype for such a system, called Transportable English Datamanager (TED) was implemented at SRI International [Hendrix81a]. TED differs from SHRDLU in that a formal data base

structure is used. TED incorporates an "automated interface expert" to interview the user about the structure and meaning of an individual data base. It then maps the representation of a parsed query onto the given data base structure. The DBMS forms an integral part of the TED system, allowing the data acquisition and retrieval process to be integrated with the parsing process. TED's architecture is outlined in Figure 2.

A system administrator must first supply information about the target data base to TED's automated interface expert. The expert then stores these facts in TED's specialized data base management system. The parser can then accept information from the user, similar to the method used by most commercial data base management systems. (Note that TED accepts natural language queries, but not natural language adds or updates to information in the data base.) The parser accepts natural language queries from the user and converts these queries into formal queries which match the structure of the target data base. The built-in data manager then retrieves the specified information and displays the information to the user.

TED's architecture deals with the problem of data domain

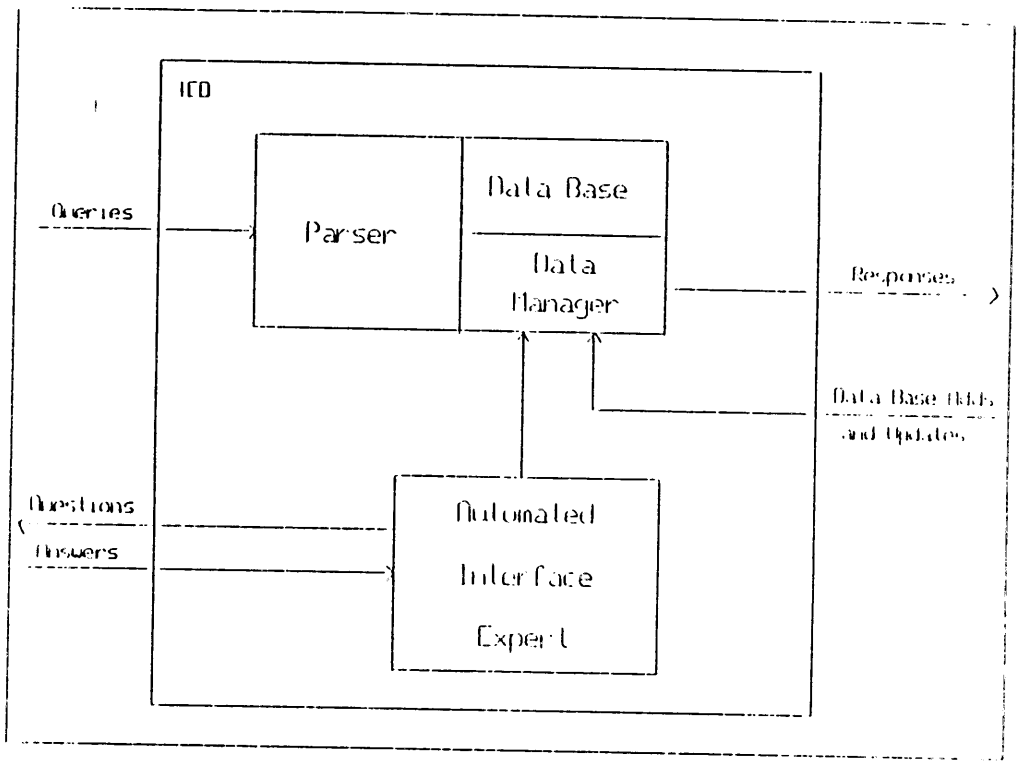


Figure 2: Architecture of TED

dependency, because it accepts background information from the system administrator, rather than having such information pre-programmed into the parser/interpreter. Although TED can interpret an acceptable subset of natural language, it does not have the richness of understanding that LADDER has, because all contextual information must be supplied to it by the system administrator. However, the effort saved in transporting the system to a new data domain more than offsets the limits in its syntactic and semantic coverage.

Despite transportability across data domains, the TED architecture inhibits the transportability of the system to other DBMS's. The purpose of the TED prototype is to develop a system which is transportable across different domains of data (data base definitions), rather than across different data management systems.

As a prototype, the TED system accomplishes its goals. The user defines a new data base file structure, then TED interviews the user about the structure and semantics of the new file. The system can then accept queries in English about the new file, and return proper answers. Because TED is a prototype, it is designed with only limited syntactic coverage. For instance, the only verbs understood by TED are forms of BE and HAVE. The

designers of TED contend that certain extensions can be made to the system, such as expanding the syntactic coverage of the parser and adding a frame structure to interpret forms of other verbs, in order bring TED's coverage to a commercially acceptable level.

2.3. FULLY TRANSPORTABLE NATURAL LANGUAGE INTERFACES

Recently, natural language interfaces have been developed which offer both data domain independence and data-manager independence. Two strategies have been used to attain this transportability. The first, used in systems such as IRUS [Bates86] and Datalog [Hafner85], attempts to separate the semantic processing routines into modules, some of which are applicable to any data domain, and some of which are specific to a given domain. When transporting the natural language understander to a new domain, many of these semantic modules can be reused. This strategy assumes that transportability to a different data domain is to be handled as a special case, rather than as the norm.

The second strategy, used by systems like TED, ASK [Thompson85], and TEAM [Grosz87], allows the domain-specific information to be gathered by an "interface

expert", then stored as information in a knowledge base. No reprogramming is required. Whichever method is used, for full transportability the result of the parse must be an intermediate query language which can be converted to many common data base query languages using only syntactic methods. This is because the syntax of the target query language is not known in advance.

As an example of the first strategy above, we will now look at the architecture of Datalog. Its structure is shown in Figure 3. The user submits queries to Datalog's parser, which is based on a cascaded augmented transition network (ATN) parser. The ATN uses syntactic modules, semantic modules which can be used in any domain, and semantic modules to be used in a specific domain. Likewise, the lexicon is separated into lexical terms which can be used in any domain, and lexical terms which are to be used in a specific domain. The parser then produces an intermediate query language, which can be converted to the DBMS query language using a simple syntactic mapping.

In order to transport Datalog to a new domain, additional domain-specific semantic modules must be written, and the domain-specific lexicon must be extended. Therefore,

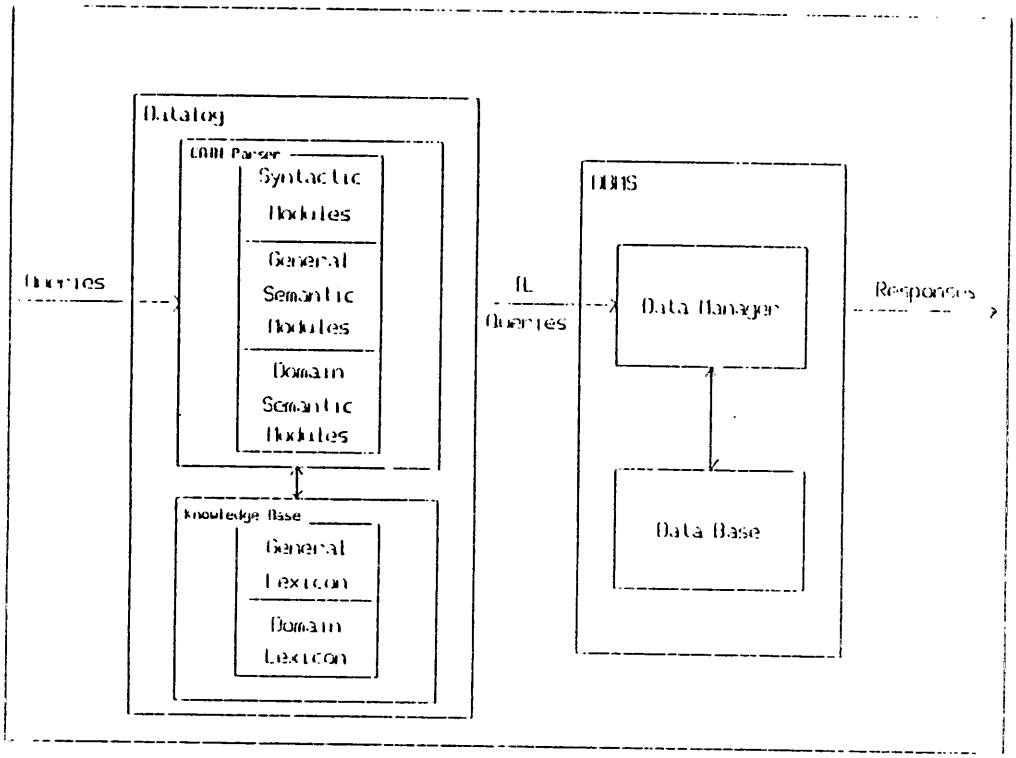


Figure 3: Architecture of Datalog

although it is much easier to transport Datalog to a new domain than earlier systems, it still requires additional programming.

As an example of the alternative strategy, we will now look at the architecture of TEAM. Its structure is shown in Figure 4. Like TED, TEAM uses an automated interface expert to interview the system administrator about the structure and meaning of the target data bases. But instead of storing this information in a custom data manager, TEAM builds its own knowledge base. The knowledge base consists of three parts. The first part is a conceptual schema, which contains information about the objects referred to in the data bases and their properties and interrelationships. The second part is a data base schema, which maps these objects and relationships into the target data base structures. The third part is a lexicon, which maps these objects and relationships into the words and phrases used to refer to them.

Once the knowledge base is defined for the target data base, a user can query the data base using the words and phrases defined for TEAM by the system administrator. The parser, referencing the knowledge base, translates the English-language query into an intermediate-language

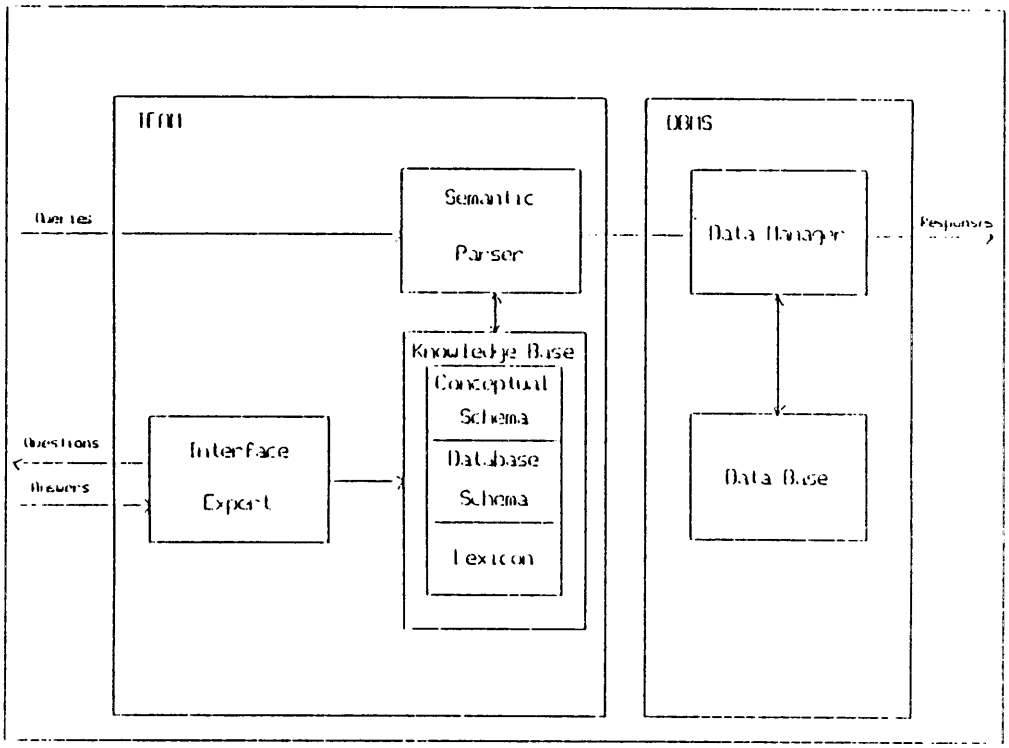


Figure 4: Architecture of TEAM

query called a SODA query [Moore79]. The SODA query differs from the natural-language query in that it is a well-defined, unambiguous representation of the query. Any data base management system can then convert the SODA query into its own query language using a simple syntactic mapping.

TEAM suffers from the same lack of understanding as TED does, because it also must rely on the system administrator to supply all domain-specific contextual information. Existing transportable systems provide a meaningful interface, but are only the first steps toward successful and useful transportable natural language interfaces.

3. AN ARCHITECTURE FOR TRANSPORTABILITY

The approach taken in this thesis is structured in much the same way as the approach taken in TEAM. Since this approach was developed independent of the work on TEAM (the main results of the TEAM project were published in May, 1987), it is interesting to note the common architecture and philosophy. The architecture proposed is shown in Figure 5.

Like TEAM and TED, this architecture incorporates an interface expert to acquire information about the structure and meaning of the target data base. This information is stored in a separate knowledge base. The parser uses this knowledge base to parse English-language queries into an intermediate-language query The intermediate-language query is an unambiguous representation of the query in the structure of the target data base(s) which can be converted into the target DBMS's query language using a simple syntactic mapping.

The major difference between the architecture of the proposed system and TEAM is that TEAM parses the natural language query into a parse tree, then into logical forms (an extension of first-order logic). TEAM then uses a

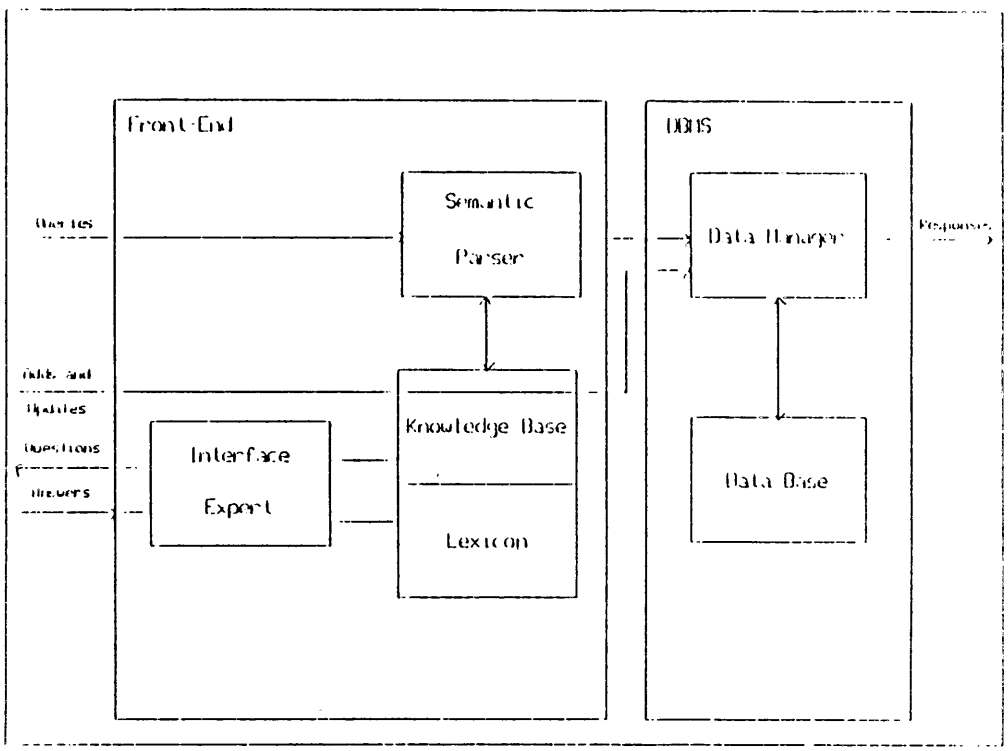


Figure 5: Proposed Architecture

schema translator to convert the logical forms into SODA queries. The proposed system uses a semantic parser to convert queries directly into intermediate query language commands.

Since the proposed architecture is similar to TEAM's, it suffers from some of the same difficult problems. The following sections detail these problems, the reason why these problems are difficult, and what can be done to solve them.

3.1. ISSUES IN THE DESIGN OF A FRONT-END SYSTEM

Data base management systems can accept information in a predefined format, update information, and retrieve information in response to very narrowly defined queries. Advanced DBMS's can also perform rudimentary, generally unsophisticated operations on the data which are stored in the data base. These operations usually summarize or apply mathematical operations to data, rather than make new inferences from existing data. The data base query language which is used to add, update, or retrieve information must be a tightly structured, restricted, and unambiguous subset of the English language. There are at least three good reasons for this.

First, natural language is often ambiguous. In order for the computer to resolve ambiguity, it must be able to use common-sense reasoning within a given domain. Current DBMS's simply do not have this capability. Despite intense research, surprisingly little progress has been made in this area, which is taken so much for granted in human reasoning.

Second, if the user wants to retrieve information in a way which is not exactly the way it is organized in the data base, he must specify the exact operations required to extract that information. A typical data management system would not know how to organize a formal query based on the question, "Where is the enemy most likely to attack?", even if it has all of the pertinent information, unless the query is broken down into simpler queries which match the structure in which the information is stored.

Finally, a narrow and precise subset of English is much easier to translate into operations that the data base manager can comprehend. A narrow and unambiguous subset of the language avoids problems such as having to recognize many different ways of asking the same question and having to determine the deep meaning of what is being communicated (semantics), given the structure of what is

being said (syntax). Also, if the system cannot perform a certain operation, the language is restricted so that it is impossible to ask that the operation be done. In this way, the system never receives a request which it cannot at least attempt to fulfill.

Deductive data base management systems employ advanced knowledge representation and deductive reasoning techniques. However, such systems are only a research area, and no deductive DBMS has gained a measure of wide commercial use. Since the object of this paper is to provide a natural language interface to existing DBMS's, the capabilities of deductive data managers are not considered.

Despite the problems which arise as a result of accepting English language queries to data bases, progress has been made. But the user of such a system may be disappointed by the system's inability to answer complex questions, to understand the meaning of what is being said, or to draw new inferences from existing facts. Also, it is very difficult to apply these systems to a new data domain.

In designing a natural language front-end to a DBMS, the above limitations inherent in DBMS's causes corresponding

problems in a natural language understanding system. In the development of a natural language interface to a DBMS, the following limitations should be recognized.

First, the language that is accepted by the system must be a subset of the English language. The trait which sets natural language systems apart from more formal query languages lies in their ability to accept queries in many of the ways in which a user would formulate questions. This means that a natural language system must deal with the problems of ambiguity, semantics, and pragmatics.

Next, the system must be able to map the semantic interpretation of a query onto the strict format and restricted operations found in the DBMS query language. This places heavy burdens on the natural language processor, as the operations found in existing DBMS's are woefully inadequate to handle a query which is not logically organized in the same way as the data base. The system must reformulate a request for a complex operation into a series of simpler operations that the DBMS can perform. Existing natural language processing systems do a poor job in these areas, compared to the ability of humans to accomplish the same task. In addition, complex mappings are very difficult to define

with any measure of general applicability in transportable systems.

Also, since a typical DBMS cannot draw new inferences from existing information, certain queries cannot be answered even if sufficient information is available and the query is understood. This problem could be dealt with by providing an inference system between the natural language processor and the DBMS. However, existing inference systems must rely on highly domain-specific information, defeating the ultimate goal of transportability. A compromise solution is to recognize certain types of simple inferences as a set of operations that the DBMS can perform, and formulate the series of operations required. The lack of a general inference system proves to be a tremendous drawback in the design of a transportable natural language processor. Perhaps advances in knowledge representation theory will reveal an better answer in the future. For a broader discussion of these problems, see [Hendrix81b].

The architecture presented in this thesis addresses some of these hard problems. The system first collects information about the structure and meaning of its target data bases, and places this information into the

knowledge base. The semantic parser can then reason about the target domain. This allows the user to phrase queries in a way in which he is accustomed. The parser draws on the information in the knowledge base in order to resolve ambiguity and reorganize the query into a structure which matches the structure of the target data base. Therefore, the user need not phrase the question in a particular fashion in order to properly extract information from the data base.

3.2. ISSUES IN THE DESIGN OF A TRANSPORTABLE SYSTEM

A large variety of DBMS's exist today, most of which do not offer a natural language interface. Ideally, a natural language front-end should be able to interface to any of these systems with a minimum of effort. However, no natural language interface can completely divorce itself from the details of a particular data base design or subject area. For instance, ambiguity cannot be resolved without recognizing the context in which the query was asked. For instance, the ambiguity in the query "Who strikes most often?" cannot be resolved unless one knows what type of information the particular data base deals with, whether it be labor unions, bowling

leagues, baseball players, bombing squadrons, or terrorist organizations.

A natural language interface is also bound by the types of operations that the target DBMS can perform. This would restrict the types of queries that could be successfully processed. For instance, the query "How many departments have more than 100 employees?" requires the DBMS not only to retrieve the records of qualifying departments, but also to total the number of department entries which qualify. If no function exists which returns the total number of qualifying records, then this query could not be answered directly. In many cases, a simple post-processor can be implemented to provide common "missing" operations.

Although it is impossible to separate the natural language processor from the target data base completely, this is not grounds for abandoning the idea of transportability altogether. The key is to first identify the knowledge requirements in order to map an English sentence onto any target data base. These requirements can then be formalized in the design of a knowledge base. A target query language must then be defined which can be transformed into many common DBMS query languages using only syntactic methods. Finally,

strategies must be formulated to map the natural language query into the target query language referencing the information gathered in the knowledge base. Although these strategies would not by themselves be sufficient to process a wide variety of English queries, they form a nucleus around which a more sophisticated natural language processor can be built. When transporting the processor to another data base, this nucleus remains unchanged, reducing the effort required.

When discussing portability, two issues must be considered. First, the system must be portable across different data domains. This means that a portable system must operate on any type of data which can be placed into the target DBMS. For instance, a portable natural language processing system that interfaces with a data base of ship information must be easily transportable to another data base consisting of, say, personnel information. The second aspect of portability is transportability to different DBMS's. This introduces the added difficulty of not knowing in advance what the syntax of the target query language looks is, and what operation it is capable of.

This thesis does not try to solve all of the above problems, because many of them are a direct result of the limitations of current DBMS's. When DBMS's can perform complex operations on data and draw new inferences from existing information, these problems will be much easier to solve. The sciences of natural language processing and knowledge representation must work together in order to develop a system which has such capabilities. The purpose of this theis is to propose a methodology for transporting a natural language processing capability to existing DBMS's, so we are bound by existing DBMS limitations.

4. THE KNOWLEDGE BASE

4.1. KNOWLEDGE REQUIREMENTS

In order for a natural language front-end to be transportable, a knowledge base must be maintained within the natural language processor. This knowledge base must be small in comparison to the actual data bases, but it must contain all of the information required to successfully parse a query relating to the target data bases. Strategies can then be developed which use the information stored in the knowledge base.

Herein lies the most difficult problem in the design of natural language systems: What knowledge is required in order to intelligently map a natural language query into a formal request for information from a data base? As a minimum, the knowledge base must contain the following information:

1. The data definitions must be specified, including the data file names, field names, and field types.

Without a description of how the data is organized in the data base, it would be impossible for the natural language processor to organize its output so that it is directly translatable into a data base query. This is

because the data could be organized in various ways. For instance, consider the query, "How many employees are in the accounting department?". To solve this query, the DBMS may have to look in a file of department information, locate the record associated with the accounting department , then return the value in a NUMBER-OF-EMPLOYEES field of that record. Alternatively, the DBMS may have to look in a file of employees, then count the number of records that have a value of "ACCOUNTING" in a DEPARTMENT-NAME field.

2. Corresponding synonyms and verbs for files and fields, and corresponding adjectives and units of measure for numeric fields must be specified.

Corresponding synonyms are English-language equivalents for file and field names. Corresponding verbs describe an action which defines the action implied by a particular file or field in the data base. For instance, WORK may be a corresponding verb for a data base of employees, because by definition, all employees work. Corresponding adjectives and units of measure define attributes of a numeric field. For instance, a field which contains the age of an employee is defined by a corresponding adjective OLD. The unit of measure is YEAR. Once this information is gathered and placed into

the knowledge base, strategies can be developed to map English-language phrases into corresponding data base files and fields.

3. Key-value fields must be identified for each data file.

Key values are typically used as retrieval values. For instance, any query which starts "Which ships..." indicates that the user is asking for the key values of a ship file, whether it be ship name, identification number, or other unique identifier.

4. The links between data base files must be specified.

The way data base files relate to one another is useful in understanding a query which requires information from more than one data base file. For example, let's assume that two data base files exist. The first contains individual ship information such as class, age, cargo, etc. The second lists ship classes and attributes common to each class of ship, such as speed, size, etc. Because both files possess a field which has a common domain (ship class), the two files can be JOINed [Date81], resulting in a single data base file with each ship

inheriting the attributes of its class. If a query then asks, "What is the speed of the CVN-72," the system can respond by recognizing that the CVN-72 is of a certain class (aircraft carrier), and retrieving the speed of aircraft carriers in the ship class data base file.

5. The identity and contents of each field which modifies the data base file must be known.

A modifying field is a field whose contents serve to describe, define, or classify the object of the data base file. For instance, in a file of ships, SHIPTYPE is a modifying field, because the contents of this field serves to classify a particular ship. On the other hand, PROPELLER-TYPE is not a modifying field, since it classifies a part on the ship (the propeller), rather than the ship itself. If modifying information is stored in the knowledge base, then given the query, "How many aircraft carriers are in the Med?," the parser can recognize that "aircraft carrier" is a type of ship, and qualify the formal query accordingly. Similarly, it can recognize that in one context, "Nimitz" is a person, and in another context, "(the) Nimitz" is a ship.

The above information is supplied to the natural language processor either by the interface expert interviewing the

system administrator or by monitoring data base adds and updates as they occur. Information falling under the first four headings above is fairly static, as it generally only changes when the data definitions change. Therefore, this information can be supplied by the system administrator via the interface expert. But information under the fifth heading may be quite volatile, since it may change any time a change is made to data in modifying fields in a data base. Therefore, the natural language processor should monitor adds and updates made to the data base to note any changes made to these fields, and change its own knowledge base accordingly.

Further customization to the target data base can reduce unsolved ambiguities and other inadequacies left by failures in strategies which can be formalized using the above types of information. It may also be necessary to add highly specialized contextual information, which only applies to a particular data base, in order to increase the scope of what is recognized by the parser to an acceptable level. Finally, if the target DBMS is capable of atypical operations, such as mathematical computations or deductive processes, customization can enable these operations to be used. The customization process is intended to be a minor task, but may be extensive in

certain cases. The intent of developing a transportable front-end processor is to provide a substantive core around which a more sophisticated customized processor may be built. Of course, the system should be capable of handling many straight-forward queries without any customization.

4.2. USING A SEMANTIC NETWORK

The example knowledge base utilizes a semantic network to store the information required in part 4.1. A semantic network is not the only way to represent this information-- indeed any number of knowledge representation schemes are acceptable. However, there are advantages of using a semantic network.

First, a semantic network stores information in a structure which is convenient to use when attempting to resolve ambiguity. For instance, in determining whether "Nimitz" is a person or a ship, the ambiguity can be resolved simply by determining which of the structures shown in Figure 6 are present in the semantic network.

If both structures exist in the network, then the system is at least able to narrow the meaning of "Nimitz" to either a person or a ship. Then the system can take

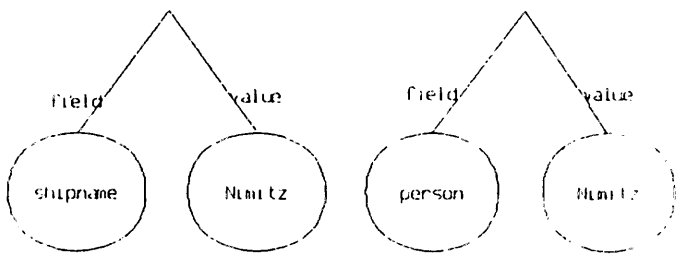


Figure 6: Network Entries for "Nimitz"

steps to determine which of the two the reader is referring to, either by looking at the name in context or by asking the user directly.

Second, a semantic network allows for fairly simple customization of the knowledge base, through expanding the network to include additional information about each field, and how certain words and phrases are to be interpreted. This information can be added to the network without changing the original design of the natural language processor. Since some customization will typically be necessary, this is a desirable feature.

Finally, a semantic network is compatible with the integration of an inference engine, which has the ability to infer new facts based on existing information. In this way common sense rules can be added to the network, and the natural language processor can use these rules to parse the query correctly.

The structure of the semantic network which represents the example knowledge base is defined in appendix A.

5. THE INTERFACE EXPERT

5.1. MAPPING NATURAL LANGUAGE ONTO A DATA BASE STRUCTURE

Now that knowledge base requirements are defined, a method must be developed to easily add the domain-specific knowledge into the knowledge base. A common method (TED, TEAM) is to use an "interface expert". The interface expert is a program which queries the administrator of the data base management system about the target data bases. This interviewing process occurs only once, at the time when a new data base is introduced to the system. The information acquired is stored in the natural language processor's own knowledge base. Afterwards, a user can query the data base, and the natural language system will refer to the description of the data collected by the interface expert.

The approach proposed in this thesis requires the interface expert to acquire a mapping from English words (lexical terms) to the data base files, fields, and values. In order for the parser to use this mapping, it must be broken down into various parts of speech and rigidly defined. The remainder of this chapter is an attempt to define such a mapping.

A data base field is typically one of three types:

numeric, alphanumeric, or boolean. Each of these types indicates different kinds of values. Numeric fields usually indicate amount, scale, or degree. Boolean fields usually indicate whether or not an entity has a certain property. Alphanumeric fields, although not well defined for our purposes, frequently indicate a classification of an item. These traits imply that numeric fields map well into adjectives, comparatives, and superlatives (old, heavy, higher, best). Boolean fields map well into verb phrases and adjectives (have missiles, {is} male, {is} injured) and modifiers (docked, married). Alphanumeric fields map well into nouns (ship class, employee name). These are broad categories, however, and more formalization is needed in order to make use of these intuitive patterns. Let's look at each part of speech individually.

5.2. MODIFIERS

In noun-noun or adjective-noun phrases, the first noun(s) or adjective(s) usually (though not always) modifies the last noun. Similarly, certain fields in a database serve to modify the subject of the file which contains that field. As indicated above, if the modifier is noun-noun, then the corresponding data base field is usually

alphanumeric. If the modifier is adjective-noun, then the corresponding data base field is usually boolean or numeric. This brings us to the first three mapping techniques.

> Strategy 1: If a noun-noun phrase is encountered, the phrase can be parsed if the last noun can be mapped to a synonym N of a data base file, and any preceding nouns can be mapped to alphanumeric fields that serve to modify N. An alphanumeric field modifies N if A is a synonym of the field, A(i) is a data element of the field, and the phrase "A(i) N" makes semantic sense.

Synonyms are defined in the usual way, except that a phrase is a synonym of itself. Synonyms for EMPLOYEES may include EMPLOYEES, PERSONNEL, and WORKERS.

Examples of alphanumeric fields which serve as modifiers:

> Field name SHIPCLASS modifies the SHIP file:
> NELSON is an element of the field SHIPCLASS, and
> NELSON SHIP makes semantic sense.
>
> Field name SEX modifies the EMPLOYEE file:
> MALE is an element of the field SEX, and
> MALE EMPLOYEE makes semantic sense.
>
> Field name DOCKED-AT does not modify the SHIP file:
> PARIS is an element of the field DOCKED-AT, but
> PARIS SHIP does not make semantic sense.

> Strategy 2: If a adjective-noun phrase is encountered, the phrase can be parsed if the noun can be mapped to a synonym N of a data base file, and the adjectives can be mapped to boolean fields that serve to modify N. A

boolean field modifies N if B is a synonym of the field, and the phrase "B N" makes semantic sense.

Examples of boolean fields which serve as modifiers:

Field name MARRIED modifies the EMPLOYEE file:
MARRIED EMPLOYEE makes semantic sense.

Field name DOCKED modifies the SHIP file:
DOCKED SHIP makes semantic sense.

Field name HAS-WARHEADS modifies the SHIP file:
ARMED is a synonym of HAS-WARHEADS, and
ARMED SHIP makes semantic sense

Strategy 3: If a adjective-noun phrase is encountered, the phrase can be parsed if the noun can be mapped to a synonym N of a data base file, and the adjectives can be mapped to numeric fields that serve to modify N. A numeric field modifies N if C is a corresponding adjective of the field, C(est) is the superlative form of C, and the phrase "C(est) N" makes semantic sense.

Corresponding adjectives serve to describe to what degree a numeric field's elements possess the trait of that field. For instance, corresponding adjectives of the field named AGE include OLD, AGED, and YOUNG. OLD and AGED are maximizing adjectives, whereas YOUNG is a minimizing adjective. Maximizing and minimizing adjectives are antonyms.

Examples of numeric fields which can be used as modifiers:

> Field name AGE modifies the EMPLOYEE file:
> OLD is a corresponding adjective of AGE, and
> OLDEST EMPLOYEE makes semantic sense
>
> Field name QUALITY modifies the RESTAURANT file:
> GOOD is a corresponding adjective of QUALITY, and
> BEST RESTAURANT makes semantic sense
>
> Field name SIBLINGS does not modify the STUDENT file:
> There is no corresponding adjective of SIBLINGS.

Note that the field name RUNWAY-LENGTH would not modify an AIRPORT file, because, if LONG is a corresponding adjective of RUNWAY-LENGTH, then LONGEST AIRPORT would not make the intended semantic sense.

Note also that the superlative form of the adjective must be used in adjective-noun phrases, otherwise the query would be vague (old employees, long ships, heavy cargo).

5.3. ADJECTIVES

Numeric fields map well to adjectives. This brings us to the next technique.

> Strategy 4: If C is a corresponding adjective for a file, and N is a synonym of the file name, then "HOW C be N" makes semantic sense.

For the corresponding adjectives found above, this means that the phrases:

HOW OLD ARE {the} EMPLOYEES?
HOW GOOD IS {the} RESTAURANT?

will make semantic sense. The next technique is similar:

> Strategy 5: If C is a corresponding adjective for a file, and K is an alphanumeric key-value field of that file, and K(i) is a data element of the field K, then "HOW C be K(i)" makes semantic sense. K is a key-value field if it uniquely qualifies the subject of the file.

From the above examples, EMPLOYEE-NAME might be the key-value field for the employee file, and RESTAURANT-NAME might be the key-value field for the restaurant file. Therefore, the phrases:

HOW OLD IS JIM SMITH?
HOW GOOD IS THE FOUR SEASONS?

will make sense, assuming that "Jim Smith" is found in the field EMPLOYEE-NAME, and "Four Seasons" is found in the field RESTAURANT-NAME.

5.4. INTRANSITIVE VERBS

Intransitive verbs appear in queries with no direct object. The subject of the data base file (denoted by the key field) is typically the agent of the act described by an intransitive verb.

> Strategy 6: If V is a corresponding verb for the subject S of a data file, then V can be used as the verb in a query in which the subject is S and no direct object exists.

Note that nearly all data files have the corresponding verb EXIST, because subjects of data files are real.

Corresponding verbs for data files must be intransitive. In addition, they must make the intended semantic sense when used in the following sentence:

BY DEFINITION, ALL <subject>S <corresponding-verb>.

Where <subject> is the subject of the data base file. The meaning of "by definition" is more intuitive than formal. Essentially, it means "because <subject>s <verb>, they are included in this data file." For instance, in a file of employees, a corresponding verb would be WORK, because, by definition, all employees work. "By definition" means "because employees work, they are included in the employee data file." Therefore, the following queries are acceptable:

> How many employees work in the auditing department?
> Does John Smith work in the tax department?

Note that all employees also earn, write, and drive. However, these verbs are transitive-- employees earn money, write papers, and drive vehicles-- and therefore

are not appropriate for corresponding verbs for an employee file.

Since no semantic meaning is taken from the verbs themselves, some verbs may be inappropriate as corresponding verbs for data bases. For instance, all employees perspire. However, perspiring has nothing to do with being an employee, so the meaning of what it is to perspire is lost in the parse. This is the reason for the "by definition" requirement.

Other examples of corresponding verbs to data base files:

All ships sail:
Does the U.S.S. Enterprise sail to France?

All (retail store) customers shop:
How many customers shop at department stores?

All (restaurant) customers dine:
How many customers dine at the Eat-em-up restaurant?

5.5. TRANSITIVE VERBS

Transitive verbs accept a direct object. The most common structure which uses a transitive verb is a sentence in which an agent <transitive-verb>s an object. In mapping this structure to a data base file, the agent is usually the subject of the data base file (denoted by the key

field), and the object is a modifying field of that file.

> Strategy 7: If a verb V is a corresponding verb to a numeric or alphanumeric field in database D, then that field name represents a direct object in sentences where the agent is the subject of D, and the verb is V.

For example:

> SALARY is a numeric field in the file EMPLOYEES.
> EARN is a corresponding verb for SALARY, because employees earn a salary.
>
> SPOUSE is an alphanumeric field in the file EMPLOYEES.
> MARRY is a corresponding verb for SPOUSE, because employees marry spouses.
>
> PATIENT is an alphanumeric field in the file DOCTORS.
> TREAT is a corresponding verb for PATIENT, because doctors treat patients.
>
> FATHER is an alphanumeric field in the file STUDENTS.
> FATHER has no corresponding verbs, because students don't do anything to fathers.

(Actually, students do many things to fathers, but not anything defined by the field FATHER.)

For numeric fields, the active or passive form of the verb can be used in a query. For instance:

> How many employees earn at least 15000 dollars?
> How much salary do the employees earn?
> How much salary is earned by the employees?
>
> Does the Eat restaurant serve more than 140 customers?
> How many customers does the Eat restaurant serve?
> How many customers are served by the Eat restaurant?

Likewise for alphanumeric fields:

Who did John Smith marry?
Who is married to John Smith?
Who is John Smith married to?

Who do the doctors treat?
Who is treated by the doctors?
Who are the doctors treating?

What did the chefs cook?
What was cooked by the chefs?

Strategy 8: If a verb V is the past participle of a corresponding verb to a boolean field of a data base file, and O is the object of the data base file, then either "O is V" or "O has V" describes the meaning of the field.

For example:

HAS-SPOUSE is a boolean field in the file EMPLOYEES
MARRY is a corresponding verb to HAS-SPOUSE:
"employee is married" means HAS-SPOUSE.

DETONATED is a boolean field in the file BOMBS
EXPLODE is a corresponding verb to DETONATED:
"bomb has exploded" means {has} DETONATED.

Examples of queries based on these corresponding verbs:

Is Joe Short married?
How many employees are married?
Has the bomb exploded?
How many bombs have exploded?

5.6. UNITS OF MEASURE

Numeric fields have units of measure associated with

them. For instance, runway length might be measured in feet, age in years, or salary in dollars. Such units of measure are useful in parsing sentences such as:

How many years old is Jim Smith?
What ships are longer than 150 feet?
How many dollars does Jim Smith earn in a year?

Units of measure are perhaps even more useful in generating responses to queries:

How old is Jim Smith? 45 years.
How long is the longest ship? 415 feet.
How much does Jim Smith earn per year? 28000 dollars.

Strategy 9: If <units> defines a proper unit of measure for a numeric field, then the following phrases have semantic meaning in the reference to the field:

<units>
<units> of <numeric-field>
<units> <corresponding-adjective>

<numeric-field> refers to a numeric field which includes <units> as a proper unit of measure. <corresponding-adjective> refers to a corresponding adjective referring to such a numeric field.

Examples of using units of measure follow:

"years" is a unit of measure for field name AGE, so
"years"
"years of age"
"years old"
all have semantic meaning in reference to AGE.

"dollars" is a unit of measure for SALARY, so
"dollars"
"dollars of salary"
"dollars wealthy"
all have semantic meaning in reference to SALARY.

"feet" is a unit of measure for LENGTH, so
"feet"
"feet of length"
"feet long"
all have semantic meaning in reference to LENGTH.

It is a simple task for the parser to recognize like units of measure, such as inches for feet or grams for tons. For instance, if "miles" is a proper unit of measure for LENGTH, then so should "millimeters" be a proper unit of measure. This means that constants should be maintained to convert the answer retrieved from the data base into the requested unit of measure.

6. THE INTERMEDIATE QUERY LANGUAGE

6.1. PURPOSE AND REQUIREMENTS

Ideally, the output of the natural language processor would be a query acceptable to the DBMS, the answer to which is also the answer to the English language query. However, when designing a portable front-end natural language processor, the syntax of the target query language is not known in advance. In order to circumvent this lack of information, the output of the natural language processor must be intermediate query language commands. The intermediate query language should be easily convertible by a post-processor into a wide range of target data base languages using only syntactic methods. Therefore, an intermediate query language must have the following traits:

1. It must contain no ambiguity.
2. It must specify that the data be retrieved as it is organized in the target data base.
3. It must not require operations on the data that the DBMS cannot perform. Since we do not know beforehand which operations the target DBMS is capable of handling, we restrict ourselves to operations that are commonly included in a DBMS, or that we can reasonably expect a post-processor to the DBMS to handle.

6.2. EXAMPLE INTERMEDIATE QUERY LANGUAGE

The intermediate query language used by the example system consists of simple expressions which indicate what is to be retrieved, what qualifications are needed, what operations are required on the retrieved data, and what links are required between data base files. These expressions are contained in four distinct registers:

The RETRIEVAL register: This register contains the fields that are to be retrieved as a result of the request. In particular, it contains a list of pairs of the form <filename>.<fieldname>, where <filename> is the name of the data base file, and <fieldname> is the name of the data base field. For instance, if the original query is "What is John Smith's age and weight?", then the RETRIEVAL register might contain (EMPLOYEE.AGE EMPLOYEE.WEIGHT).

The QUALIFICATION register: This register contains the clauses which serve to qualify or restrict the subject of the query. It has the following structure:

```
register ::= ( <clause> <next-clause>* )
<next-clause> ::= <connector> <clause>
<connector> ::= AND | OR
<clause> ::= {NOT} ( <field-name> <comparator>
<value> ) |  ( <superlative> <field-name> )
<field-name> ::= filename.fieldname
```

```
<value> ::= "anystring" | integer
<comparator> ::= < | > | >= | <= | = | <>
     <superlative> ::= GREATEST | LEAST
```

Examples of valid values for the QUALIFICATION register:

```
( EMPLOYEE.NAME="John Smith" AND NOT EMPLOYEE.AGE>20 )
( SHIP.LENGTH>200 OR GREATEST SHIP.LENGTH)
```

Superlative clauses should occur last in the qualification register, because they would have the lowest precedence (be the last operation) in a conjunctive clause. For instance, in the query "Give me the smallest aircraft carrier", the DBMS must first extract all aircraft carriers from the SHIP file, then choose the smallest craft from that subset.

The OPERATION register: This register determines the operation to be performed on the retrieved fields. Valid operations include:

```
LIST: Simply list the values in the fields
      retrieved.
COUNT: Count the number of qualifying values.
TOTAL: Sum the values of all qualifying
      numeric fields.
YESNO: If any record qualifies, then return
      "yes". Otherwise, return "no".
```

The JOIN register: This register is used when more than one data base file is needed. The register specifies the links between the files so that all files can be joined

into one data structure for retrieval purposes. The JOIN register consists of pairs of <file-name>.<field-name> specifications. The file specified by the first of the pair is to be joined into the file specified by the second of the pair, using the corresponding field names. For instance, if the JOIN register is:

(SHIP-FILE.SHIP-CLASS CLASS-FILE.CLASS-NAME)

then for each record in the SHIP-FILE, values of the fields of CLASS-FILE are joined into the record if and only if the value of SHIP-FILE.SHIP-CLASS is equal to the value of CLASS-FILE.CLASS-NAME.

6.3. POST PROCESSOR REQUIREMENTS

The registers described above can be converted to almost any DBMS query language using only simple syntactic methods. The example query language used in this thesis is the Model 204 DBMS query language [Model83]. This query language is structured much like a procedural programming language, in that a complex query can be executed in several distinct steps. The flow of control is either sequential or manipulated through the commands themselves. The language includes commands that open data base files, perform retrieval and display operations

on the files, and assign values to local variables. Conditional and looping commands are also available. Still other commands are available, but these particular commands are the ones used for the translation from the intermediate language.

In order to translate the intermediate language registers into Model 204 query language, the target data base files must first be identified. This is done by scanning the RETRIEVAL and QUALIFICATION registers for all filenames. For each unique filename found, a Model 204 DBMS command is generated of the form:

<statement-number>. USE <filename>

This command opens the data base files which will be used with the query. The statement number serves as a label for the command so that other commands can reference this command if required. The first statement number is 1, the second 2, and so on.

Next, certain records in the data base file(s) may have to be extracted and used as a qualified subset of the complete data base file. This is accomplished by converting the information in the QUALIFICATION register into query commands. For phrases which do not use superlative clauses, the transformation is as follows:

```
QUALIFICATION register:
   {NOT} ( filename.fieldname <comparator> <value> )

Query language command:
   <statement-number>.   FIND ALL RECORDS IN
       <label>  FOR  WHICH  {NOT}  fieldname
          <comparator> <value>
```

The meaning of <label> depends on whether or not this is the first phrase in the QUALIFICATION register. If it is, then <label> refers to the USE statement for that data base file. Otherwise, <label> is the <statement-number> of the command generated by the prior phrase in the QUALIFICATION register which referred to <filename>.

For phrases which use superlative clauses, the transformation is as follows:

```
QUALIFICATION register:
     ( GREATEST|LEAST filename.fieldname )

Query language command if GREATEST is specified:
     <statement-number>.  %GREATEST = 0
     <statement-number>.    FOR ALL RECORDS IN <label>
          IF fieldname > %GREATEST THEN
               %GREATEST = fieldname
     <statement-number>.  FIND ALL RECORDS IN <label>
          FOR WHICH fieldname = %GREATEST

Query language command if LEAST is specified:
     <statement-number>.  %LEAST = <max-value>
     <statement-number>.  FOR EACH RECORD IN <label>
          IF fieldname < %LEAST THEN
               %LEAST = fieldname
     <statement-number>.  FIND ALL RECORDS IN <label>
               FOR WHICH fieldname = %LEAST
```

%GREATEST and %LEAST are local variables. <label> is as defined for comparative phrases. <max-value> is a numerical value at least as great as the highest possible numerical value in <fieldname>.

Once a qualifying subset of the data base is defined, an operation must be performed on it. The proper operation is defined by the OPERATION register.

Next, the specified operation must be performed on the records which qualify. For OPERATION register value of LIST, the following command is generated:

```
<statement-number>. FOR EACH RECORD IN <to-file-name>
    FOR EACH RECORD IN <from-file-name> FOR WHICH
       <to-field-name> = <from-field-name>
       PRINT <field-name> (AND <fieldname>)*
```

The <fieldname>s are extracted from the RETRIEVAL register. The values <from-file-name>, <from-field-name>, <to-file-name>, and <to-field-name> are extracted from the JOIN register. If the JOIN register is empty, then the above structure reduces to:

```
<statement-number>.  FOR EACH RECORD IN <label>
   PRINT fieldname (AND fieldname)*
```

For OPERATION register value of TOTAL, the following commands are generated:

```
<statement-number>.  %TOTAL = 0
<statement-number>.  FOR EACH RECORD IN <to-file-name>
   FOR EACH RECORD IN <from-file-name> FOR WHICH
      <to-field-name> = <from-field-name>
      %TOTAL  = %TOTAL + <field-name> (+ <field-name>)*
<statement-number>.  PRINT %TOTAL
```

%TOTAL is a local variable. All other elements are as defined previously. If the JOIN register is empty, then the above structure reduces to:

```
<statement-number>.  %TOTAL = 0
<statement-number>.  FOR EACH RECORD IN <label>
   %TOTAL = %TOTAL + fieldname (+ fieldname)*
<statement-number>.  PRINT %TOTAL
```

For OPERATION register value of COUNT, the following command is generated:

```
<statement-number>.  FOR EACH RECORD IN <from-file>
   PRINT COUNT IN <to-file> FOR WHICH
      <from-field> = <to-field>
```

If the JOIN register is empty, then the above structure reduces to:

```
<statement-number>.  PRINT COUNT IN <label>
```

For OPERATION register value of YESNO, the following command is generated:

```
<statement-number>. FOR EACH RECORD IN <to-file>
FIND ALL RECORDS IN <from-file> FOR WHICH
   <from-field> = <to-field>
```

```
<statement-number>. IF COUNT IN <label> = 0 THEN
                       PRINT "NO"
                    ELSE
                       PRINT "YES"
                    ENDIF
```

If the JOIN register is empty, then the above structure reduces to:

```
<statement-number>. IF COUNT IN <label> = 0 THEN
                         PRINT "NO"
                    ELSE
                         PRINT "YES"
                    ENDIF
```

Note that all transformations are simple and straightforward conversions from the syntax of the intermediate query language to the syntax of the DBMS language. All ambiguity and semantic interpretation is resolved before the intermediate query language statements are generated, and no reorganization is needed to map the structure of the commands onto the structure of the data base. This is the ultimate goal of the natural language processor.

Example parses used in the demonstration system developed in conjunction with this paper appear in appendix F.

7. THE PARSER

The parser used with the example natural-language processor is based on an Augmented Transition Network (ATN) [Woods73,Bates78]. The particular implementation of ATN interpreter used was developed as part of the SNePS research project at the State University of New York at Buffalo, and is implemented in LISP.

The parser is not sophisticated, but it does use the information in the knowledge base to implement the strategies outlined in chapter 5, including:

-Recognition of data in modifying fields. This allows the parser to recognize, for instance, that "carrier" denotes a class of ship.

-Use of corresponding verbs and adjectives. This allows the parser to map natural language descriptions into individual fields.

-Recognition of units of measure. This gives the parse further direction in identifying the proper fields to map to.

A description of the syntactic structures which the parser can accept, a list of valid queries, and the ATN arcs used are listed in appendix C.

8. PROPOSED EXTENSIONS TO THE SYSTEM

This paper provides a first step toward mapping natural language into a data base structure. Additional methods can be employed in order to provide a richer and a more sophisticated transition. A discussion of some of these techniques follows.

8.1. NATURAL LANGUAGE GENERATION

After the query is processed by the DBMS, it is possible to pass the result to a natural language generator to formulate an answer to the query in English. The natural language generator can use many techniques in order to formulate an English response. An easy and straight-forward technique involves using the object of the query. This technique is most effective in queries where the operation is LIST. For example:

Q. Who are the employees in the tax department?
A. The employees are...

A second technique involves extracting the unit of measure and corresponding adjective associated with the retrieved field when that field is numeric. An example of this technique is:

Q. How old is John Smith?

A. 28 years old.

Or combining the first two techniques:

Q. How old is John Smith?
A. John Smith is 28 years old.

The third technique involves converting the entire query into a statement. This is useful for queries where the operation is YESNO. For example:

Q. Are there married employees in the tax department?
A. Yes, there are married employees in the tax department.

But then, the user may be satisfied with a simple yes/no response.

These techniques are useful in order to give the "feel" of a normal conversation with the machine.

8.2. INTERACTIVE PARSING

At times during the parse it may be desirable to solicit assistance of the user in order to resolve ambiguity. This is done by recognizing potential ambiguity, then querying the user as to the correct interpretation. For instance, given the query:

HOW OLD IS JOHN F. KENNEDY?

The system may recognize ambiguity by determining that John F. Kennedy is both a ship and a person. It may then resolve the ambiguity by responding:

```
BY "JOHN F. KENNEDY", DO YOU MEAN:
   1)  JOHN F. KENNEDY THE SHIP; OR
   2)  JOHN F. KENNEDY THE MAN?
```

The user may then respond accordingly, and the query can be parsed unambiguously. A second example is the query:

```
HOW MANY CRUSHED FEATHER PILLOWS ARE IN THE OFFICE?
```

The system may recognize ambiguity by determining that "crushed" can modify either "feather" or "pillows." It may resolve the ambiguity by responding:

```
BY "CRUSHED FEATHER PILLOWS", DO YOU MEAN:
   1)  CRUSHED FEATHER; OR
   2)  CRUSHED PILLOWS?
```

An interactive parse approach is taken in the EUFID system [Templeton79].

8.3. HIERARCHICAL RELATIONS

Some data base files contain fields which can be grouped into a particular class based on a hierarchical relation. For instance, a data base file containing merchant ship information may have fields which specify the amount of

corn, beans, rice, etc. currently being carried by the ship. These fields can be grouped together into a class which represents the cargo that the ship is carrying. If this hierarchical information is acquired by the natural language processor, then it can replace all references to "cargo" with "corn or beans or rice...". When the system encounters a query like:

What cargo does the Minnow carry?

It can then respond by listing the current amounts in the data base for corn, beans, rice, etc.

8.4. CONJUNCTION

Conjunctions are particularly difficult to parse because they can connect virtually any two parallel structures in a sentence. Note the differing uses of the conjunction "and" in the following queries:

How many people attended the wedding of Dave and Sarah Phillips?
How many people attended the Phillips' and the Jones' wedding?
How many people attended the wedding and reception?
How many people attended and cried at the wedding?
How many people cried at the wedding and attended the reception?
How many women and children attended the wedding?

One solution offered by B. K. Boguraev [Boguraev85] is to extend the ATN interpreter so that, upon encountering a conjunction, it identifies the structure of the phrase most recently parsed. It then predicts that it will find a similar structure after the conjunction, and attempts to parse it. If it fails to find a similar structure, it then backtracks and identifies the structure of a larger phrase that was also most frequently parsed. This process continues until a parallel structure is found. For instance, given that the following sentence fragment was parsed:

WHO ATTENDED THE WEDDING OF DAVE AND...

The ATN determines that a noun (DAVE) was most recently parsed, and predicts that another noun follows. This would successfully parse the query:

WHO ATTENDED THE WEDDING OF DAVE AND SARAH?

If this fails, the system backtracks, determines that a prepositional phrase (OF DAVE) was also most recently parsed, and predicts that another prepositional phrase follows. This would successfully parse the query:

WHO ATTENDED THE WEDDING OF DAVE AND OF FRED?

If this fails, the system backtracks, determines that a

direct object phrase (THE WEDDING OF DAVE) was also most recently parsed, and predicts that another direct object phrase follows. This would successfully parse the query:

> WHO ATTENDED THE WEDDING OF DAVE AND THE FUNERAL OF FRED?

If this fails, the system backtracks, determines that a verb phrase (ATTENDED THE WEDDING OF DAVE) was also most recently parsed, and predicts that another verb phrase follows. This would successfully parse the query:

> WHO ATTENDED THE WEDDING OF DAVE AND DECORATED HIS CAR?

Failing that, the system backtracks, determines that an entire sentence was already parsed, and predicts that another sentence follows. This would successfully parse the query:

> WHO ATTENDED THE WEDDING OF DAVE AND WHAT WERE THEIR NAMES?

8.5. DISJUNCTION

One approach to disjunction is to map the structures in a sentence connected by the word "or" into a data base OR relation. But there are cases where the word "and" can

be mapped into a data base OR relation as well. Consider the following queries:

1. What employees are married and have children?
2. List all secretaries and typists.

The first query maps, as you might expect, into a data base AND relation:

(IS-MARRIED = TRUE) AND (NUMBER-CHILDREN > 0)

The second query, however, maps into a data base OR relation:

(JOB-CLASS = SECRETARY) OR (JOB-CLASS = TYPIST)

A general rule to determine whether the word "and" refers to conjunction or disjunction is to determine whether the two structures connected by the "and" refer to the same domain. In the first query, the two structures refer to marital status and number of children, respectively. Since the domain is different, the data base AND relation is appropriate. In the second query, the two structures both refer to job classification. Therefore, the data base OR relation is appropriate.

9. CONCLUSIONS

Designing a front-end natural language understander which is transportable to different data domains introduces difficulties. Natural language understanding typically requires detailed background information about its domain. It also requires a common-sense notion of how objects behave in a new domain. Transporting the natural language understander to a new data base management system adds the additional difficulty of not knowing the syntax or operations of the target DBMS in advance.

Until DBMS technology reaches a point where it treats its data as knowledge rather than information, the most a natural language front-end can provide is an easy-to-use and human-like way to retrieve information from the data base. It accomplishes this by accepting queries in the user's own language, resolving ambiguity and reference, and formulating a query in the DBMS's language.

A successful transportable natural language interface is developed in this thesis by defining an architecture which allows transportability across both data domains and DBMS's. Knowledge requirements are then defined for the natural language interface. This knowledge is necessary in order to develop mapping strategies from

natural language to a formal query language. An interface expert is then developed to allow the natural language understander to obtain the required knowledge by interviewing a data base administrator about the structure of the target data base and the meaning of its domain.

An intermediate query language is developed which can be easily converted into many existing data base query languages using simple syntactic methods. A parser then uses the knowledge acquired by the interface expert to apply mapping techniques which convert the natural language query into the intermediate language query. A simple post-processor can then convert the intermediate language query into the target query language for submission to the DBMS.

A more useful system can be developed by adding on to the system kernel. A knowledge base can be integrated into the parsing process without extensive modification to the existing system. This is accomplished through the system's underlying semantic network architecture.

The solution presented is not an ultimate answer to the problem of natural language understanding and question-

answering. Rather, it is a method for allowing today's data base management systems to accept requests for information in the requester's most natural method of communication.

BIBLIOGRAPHY

Bates, Madeline. "The Theory and Practice of Augmented Transition Network Grammars." In Natural Language Communication with Computers, edited by L. Bolc. Berlin: Springer-Verlag, 1978.

Bates, Madeline and Bobrow, R. "A Transportable Natural Language Interface." In Proceedings from the Sixth Annual International Conference of the Association for Computing Machinery Special Interest Group on Information Retrieval, edited by J. J. Kuehn. New York: Association of Computing Machinery, 1983.

Bates, Madeline. "Accessing a Database with a Transportable Natural Language Interface." In Proceedings from the First Conference on Artificial Intelligence Applications. Los Alamitos, CA: IEEE Computer Society, 1984(a).

Bates, M.; Moser, M. G.; and Stallard D. "The IRUS Transportable Natural-Language Database Interface." In Proceedings from the First International Workshop on Expert Database Systems, edited by L. Kerschberg. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1984(b).

Boguraev, B. K. "Recognizing Conjunctions Within the ATN Framework." In Automatic Natural Language Parsing, edited by K. Sparck Jones and Y. Wilks. Chichester, England: Ellis Horwood Limited, 1985.

Bolc, Leonard and Strzalkowski, Tomasz. "Transformation of Natural Language into Logical Formulas." In Proceedings of the Ninth International Conference on Computational Linguistics, edited by J. Horecky. Amsterdam: North Holland Publishing Company, 1982.

Chandrasekaran, B. "Artificial Intelligence-- The Past Decade." In Advances in Computers, Volume 13. New York: Academic Press, 1975.

Date, C. J. An Introduction to Database Systems. Reading, MA: Addison-Wesley, 1981.

Ginsparg, J. "A Robust Portable Natural Language Data Base Interface." In Proceedings from the Conference on Applied Natural Language Processing. Santa

Monica, CA, 1983.

Grishman, R.; Hirschman, L; and Friedman, C. "Natural Language Interfaces using Limited Semantic Information." In Proceedings of the Ninth International Conference on Computational Linguistics, edited by J. Horecky. Amsterdam: North Holland Publishing Company, 1982.

Grosz, B.; Haas, N.; Hendrix, G.; Hobbs, J.; Martin, P.; Moore, R.; Robinson, J.; and Rosenschein, S. "Dialogic-- A Core Natural-Language Processing System." In Proceedings of the Ninth International Conference on Computational Linguistics, edited by J. Horecky. Amsterdam: North Holland Publishing Company, 1982.

Grosz, B. J. "Issues in the Design of Transportable Natural Language Interfaces." In Proceedings of the Fourth Jerusalem Conference on Information Technology, edited by N. S. Prywes and A. Shore. Los Alamitos, CA: IEEE Computer Society, 1984.

Grosz, B. J.; Appelt, D. E.; Martin, P. A.; and Pereira, F. "TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces." In Artificial Intelligence 32. Elsevier Science Publishers, 1987.

Haas, Norman and Hendrix, Gary G. Machine Learning for Information Management. Menlo Park, CA: SRI International, 1981.

Hafner, Carole and Godden, Kurt. "Portability of Syntax and Semantics in Datalog." In ACM Transactions on Office Information Systems: Special Issue on Transportable Natural Language Processing, edited by Robert B. Allen. New York: Association for Computing Machinery, 1985.

Harris, Mary D. Introduction to Natural Language Processing. Reston, VA: Reston Publishing, 1985.

Hendrix, Gary G. "Human Enginerring for Applied Natural Language Processing." In Proceedings from the International Joint Conference of Artificial Intelligence 1977. Cambridge, MA: 1977.

Hendrix, Gary G. and Lewis, William H. "Transportable

Natural Language Interfaces to Databases." In Nineteenth Annual Meeting of the Association for Computational Linguistics; Proceedings of the Conference, C. Raymond Perrault, chairman. Stanford, CA: Stanford University, 1981(a).

Hendrix, Gary G. and Sacerdoti, Earl D. Natural Language Processing Part One: The Field in Perspective. Menlo Park, CA: SRI International, 1981(b).

Model 204 DBMS System Overview. Cambridge, MA: Computer Corporation of America, 1983

Moore, R. C. Handling Complex Queries in a Distributed Database, Technical Note 170, Menlo Park, CA: SRI International, 1979.

Parkison, Roger C.; Colby, Kenneth Mark; and Faught, William S., "Conventional Language Comprehension Using Integrated Pattern-Matching and Parsing." In Readings in Natural Language Processing, Barbara J. Grosz; Karen S. Jones; and Bonnie Lynn Webber, editors. Los Altos, CA: Morgan Kaufman, 1986.

Seeley, L. N.; Chapman, P.; and Leuzinger, M. A., "Experimental Database Applications Using Natural-Language Processing." In Proceedings of the IEEE Western Conference on Knowledge-Based Engineering and Expert Systems, G. S. Robinson, editor. Washington, D. C.: IEEE Computer Society, 1986.

Shapiro, Stuart C. "Generation as Parsing from a Network into a Linear String." In American Journal of Computational Linguistics, 1975.

Shapiro, Stuart C. "The SNePS Semantic Network Processing System." In Associative Networks: Representation and Use of Knowledge by Computers, Nicholas V. Findler, editor. New York: Academic Press, 1979(a).

Shapiro, Stuart C. "Generalized Augmented Transition Network Grammars for Generation from Semantic Networks." In Proceedings of the Seventeenth Annual Meeting of the Association for Computational Linguistics, 1979.

Templeton, Marjorie. "EUFID: A Friendly and Flexible Front-End for Data Management Systems." In

Seventeenth Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference, Norman K. Sondheimer, chairman. La Jolla, CA: University of California at San Diego, 1979.

Thompson, Frederick B. and Thompson, Bozena Henisz. "Practical Natural Language Processing: The REL System as Prototype." In Advances in Computers, Volume 13. New York: Academic Press, 1975.

Thompson, Frederick B. and Thompson, Bozena Henisz. "ASK is Transportable in Half a Dozen Ways." In ACM Transactions on Office Information Systems-- Special Issue on Transportable Natural Language Processing, edited by Robert B. Allen. New York: Association for Computing Machinery, 1985.

Winograd, Terry. Understanding Natural Language. New York: Academic Press, 1972.

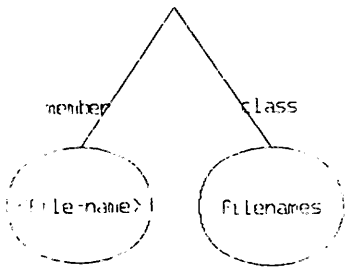
Woods, W. A. "An Experimental Parsing System for Transition Network Grammars." In Natural Language Processing, edited by Randall Rustin. New York: Algorithmics Press, 1973.

Woods, W. A. "Semantics and Quantification in Natural Language Question Answering." In Advances in Computers, Volume 17. New York: Academic Press, 1978.

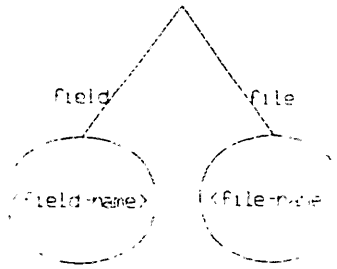
APPENDIX A: SEMANTIC NETWORK STRUCTURE

The semantic network processor used with the example was developed as part of the SNePS system [Shapiro79]. The knowledge represented enables the system to determine all of the information described in chapter 4, section A (Knowledge base requirements). The following structures comprise the network:

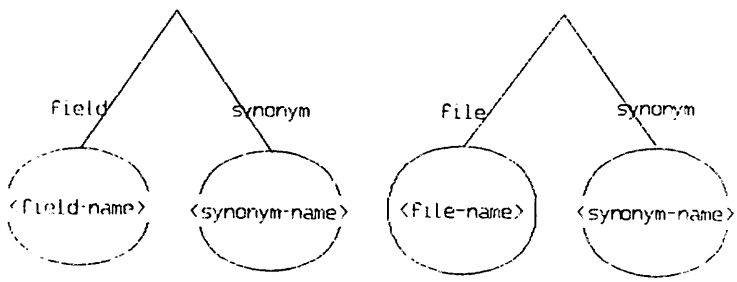
1. A name of a data base file is identified as such by relating it to the class of file names. The following structure accomplishes this:



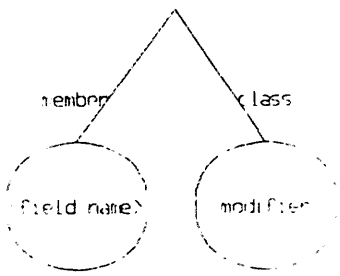
2. Each data base file consists of fields. The names of the fields contained in a particular file are related to the name of the file through the following structure:



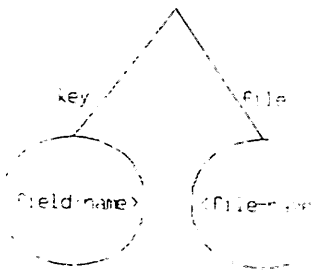
3. Both data base files and fields can have synonyms. A synonym is related to the formal name of the file or field through the following structure:



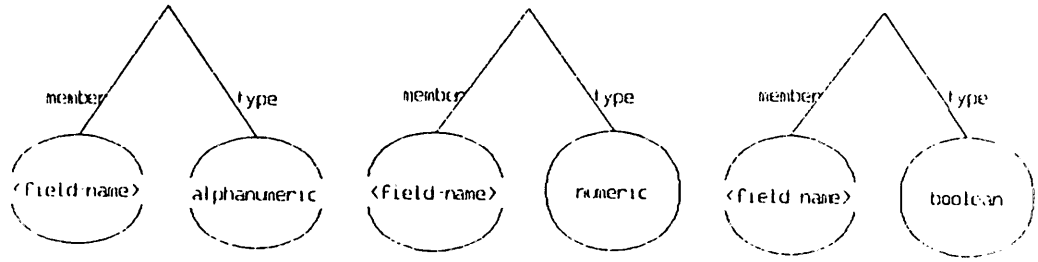
4. Certain fields must be identified as modifying fields. The network denotes this as follows:



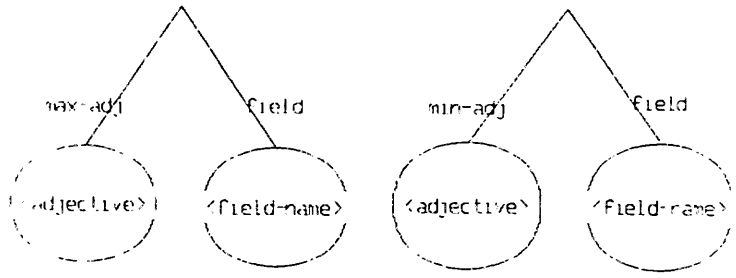
5. Certain fields are key fields. This is represented by the following structure:



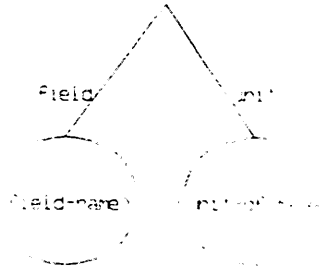
6. A field can be of type alphanumeric, numeric, or boolean. This property is represented by one of the following three structures:



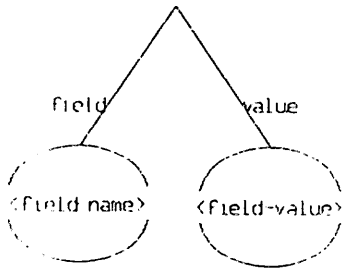
7. Numeric fields can have corresponding adjectives. These adjectives can be maximizing (old, wide, rich); or they can be minimizing (young, narrow, poor). Adjectives are related to their corresponding field names through one of the following structures:



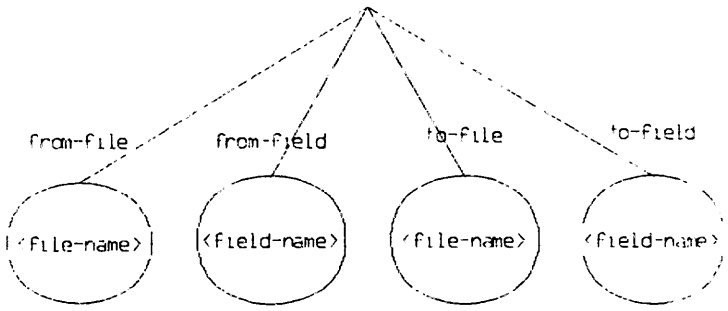
8. Numeric fields have associated units of measure. A unit of measure is related to the corresponding numeric field through the following structure:



9. Key fields and alphanumeric fields which modify the data base file have associated values. The values are related to the field name through the following structure:



10. Data base files can be linked to other files through associated fields. A link is specified in the network by the following structure:



APPENDIX B: Interface Expert Source Listing

The following is LISP source code for the interview program that collects information about the structure of the target data base by interviewing the system administrator. This code was written as part of the program written in conjunction with this paper. It stores the information collected into the SNePS semantic network using BUILD commands.

```
(member member- class class- synonym synonym- file file- key key-)
(define max-adj max-adj- min-adj min-adj- type type- field field-)
(define value value- unit unit- trans trans- intrans intrans-)
(define from-file from-file- from-field from-field-)
(define to-file to-file- to-field to-field-)

(defun interview ()
   (princ "Enter new database filename or end: ")
   (cond
      ((process-file (read-atom)) (interview))
      (t (add-links))))

(defun process-file (filename)
   (cond
      ((equal filename 'end) nil)
      (t
         (build member (^(echo filename)) class filenames)
         (definewords (cons (last-word filename) '(((ctgy . n)))))
         (princ "Done!  ")
         (process-file-synonyms filename)
         (process-intrans-verbs filename)
         (process-fields filename))))

(defun process-file-synonyms (root)
   (princ "Enter synonym for ")
   (princ root)
   (princ " or end: ")
   (cond
      ((process-file-synonym root (read-atom)) (process-file-synonyms root))
      (t)))
```

```
(defun process-file-synonym (root synonym-word)
   (cond
      ((equal synonym-word 'end) nil)
      (t
         (build file (^(echo root)) synonym (^(echo synonym-word)))
         (definewords (cons (last-word synonym-word) '(((ctgy . n)))))
         (princ "Done!  ")
         t)))

(defun process-intrans-verbs (filename)
   (princ "Enter corresponding verb for ")
   (princ filename)
   (princ " or end: ")
   (cond
      ((process-intrans-verb filename (read-atom))
         (process-intrans-verbs filename))
      (t)))

(defun process-intrans-verb (filename verb)
   (cond
      ((equal verb 'end) nil)
      (t
         (build file (^(echo filename)) intrans (^(echo verb)))
         (definewords (cons (last-word verb) '(((ctgy . v) (trans . nil)))))
         (princ "Done!  ")
         t)))

(defun process-fields (filename)
   (princ "Enter field name or end: ")
   (cond
      ((process-field filename (read-atom)) (process-fields filename))
      (t)))

(defun process-field (filename fieldname)
   (cond
      ((equal fieldname 'end) nil)
      (t
         (build member (^(echo fieldname)) file (^(echo filename)))
         (definewords (cons (last-word fieldname) '(((ctgy . n)))))
         (princ "Done!  ")
         (process-field-synonyms fieldname)
         (princ "Is this field a key field? ")
         (process-key (read-atom) filename fieldname)
         (process-trans-verbs fieldname)
         (princ "Is field (A)lpha, (N)umeric, or (B)oolean? ")
         (process-type (read-atom) fieldname))))

(defun process-trans-verbs (fieldname)
```

```
   (princ "Enter corresponding verb for ")
   (princ fieldname)
   (princ " or end: ")
   (cond
      ((process-trans-verb fieldname (read-atom))
         (process-trans-verbs fieldname))
      (t)))

(defun process-trans-verb (fieldname verb)
   (cond
      ((equal verb 'end) nil)
      (t
         (build field (^(echo fieldname)) trans (^(echo verb)))
         (definewords (cons (last-word verb) '(((ctgy . v) (trans . t)))))
         (princ "Done!  ")
         t)))

(defun process-key (yes-no filename fieldname)
   (cond
      ((equal yes-no 'y)
         (build key (^(echo fieldname)) file (^(echo filename)))
         (princ "Done!  ")
         t)
      (t)))

(defun process-field-synonyms (root)
   (princ "Enter synonym for ")
   (princ root)
   (princ " or end: ")
   (cond
      ((process-field-synonym root (read-atom)) (process-field-synonyms root))
      (t)))

(defun process-field-synonym (root synonym-word)
   (cond
      ((equal synonym-word 'end) nil)
      (t
         (build field (^(echo root)) synonym (^(echo synonym-word)))
         (definewords (cons (last-word synonym-word) '(((ctgy . n)))))
         (princ "Done!  ")
         t)))

(defun process-type (fieldtype fieldname)
   (cond
      ((equal fieldtype 'a) (process-alpha fieldname))
      ((equal fieldtype 'n) (process-numeric fieldname))
      ((equal fieldtype 'b) (process-boolean fieldname))
      (t
         (princ "Invalid response— enter A, N, or B: ")
```

```
        (process-type (read-atom) fieldname))))

(defun process-alpha (fieldname)
   (build member (^(echo fieldname)) type alphanumeric)
   (princ "Done!  ")
   (princ "Does this field modify the data base file? (Y/N) ")
   (process-modifier (read-atom) fieldname)
   (process-values fieldname)
   t)

(defun process-numeric (fieldname)
   (build member (^(echo fieldname)) type numeric)
   (princ "Done!  ")
   (process-units fieldname)
   (process-max-adjs fieldname)
   (process-min-adjs fieldname))

(defun process-boolean (fieldname)
   (build member (^(echo fieldname)) type boolean)
   (princ "Done!  ")
   (princ "Does this field modify the data base file? (Y/N) ")
   (process-modifier (read-atom) fieldname))

(defun process-modifier (yes-no fieldname)
   (cond
      ((equal yes-no 'y)
         (build member (^(echo fieldname)) class modifier)
         (princ "Done!  ")
         t)
      (t)))

(defun process-units (fieldname)
   (princ "Enter the name a proper unit of measure or end: ")
   (cond
      ((process-unit fieldname (read-atom)) (process-units fieldname))
      (t)))

(defun process-unit (fieldname unit-of-measure)
   (cond
      ((equal unit-of-measure 'end) nil)
      (t
         (build field (^(echo fieldname)) unit (^(echo unit-of-measure)))
         (definewords (cons (last-word unit-of-measure) '(((ctgy . n)))))
         (princ "Done!  ")
         t)))

(defun process-max-adjs (fieldname)
   (princ "Enter the name of a maximizing adjective or end: ")
   (cond
```

```
      ((process-max-adj fieldname (read-atom)) (process-max-adjs fieldname))
      (t)))

(defun process-max-adj (fieldname adjective)
   (cond
      ((equal adjective 'end) nil)
      (t
         (build field (^(echo fieldname)) max-adj (^(echo adjective)))
         (definewords (cons (last-word adjective) '(((ctgy . adj)))))
         (princ "Done!  ")
        t)))

(defun process-min-adjs (fieldname)
   (princ "Enter the name of a minimizing adjective or end: ")
   (cond
      ((process-min-adj fieldname (read-atom)) (process-min-adjs fieldname))
      (t)))

(defun process-min-adj (fieldname adjective)
   (cond
      ((equal adjective 'end) nil)
      (t
         (build field (^(echo fieldname)) min-adj (^(echo adjective)))
         (definewords (cons (last-word adjective) '(((ctgy . adj)))))
         (princ "Done!  ")
        t)))

(defun process-values (fieldname)
   (princ "Enter data value for field or end: ")
   (cond
      ((process-value fieldname (read-atom)) (process-values fieldname))
      (t)))

(defun process-value (fieldname datavalue)
   (cond
      ((equal datavalue 'end) nil)
      (t
         (build field (^(echo fieldname)) value (^(echo datavalue)))
         (definewords (cons (last-word datavalue) '(((ctgy . n)))))
         (princ "Done!  ")
        t)))

;the next 5 functions add links between the defined data base files
(defun add-links ()
   (princ "Enter origin file of link or end: ")
   (cond
      ((add-link (read-atom)) (add-links))
      (t)))
```

```
(defun add-link (fromfile)
   (cond
      ((equal fromfile 'end) nil)
      (t
         (princ "Enter origin field of link: ")
         (add-to-links fromfile (read-atom)))))

(defun add-to-links (fromfile fromfield)
   (princ "Enter destination file of link: ")
   (add-to-link fromfile fromfield (read-atom)))

(defun add-to-link (fromfile fromfield tofile)
   (princ "Enter destination field of link: ")
   (add-net fromfile fromfield tofile (read-atom)))

(defun add-net (fromfile fromfield tofile tofield)
   (build from-file (^(echo fromfile)) from-field (^(echo fromfield))
          to-file (^(echo tofile)) to-field (^(echo tofield)))
   (princ "Done! "))
```

APPENDIX C: THE GRAMMAR

The parser developed in conjunction with this paper parses a rudimentary grammar sufficient for demonstration purposes. The grammar accepts queries in the following forms:

1. HOW <units> BE <object> <pp>*
2. <action> BE|HAS <boolean>|<verb-intrans> <pp>*
3. <action> <pp>* EXIST <pp>*
4. <request> {THE <number> OF} <object> <pp>*
5. BE THERE {any} <object> <pp>*
6. HOW MANY|MUCH <units> {OF <numeric>} BE|DO {<object>} <verb-trans> <pp>*

<units>	<corr-adj> | MANY <unit-of-measure> OF <numeric> | MANY <unit-of-measure> <corr-adj>
<corr-adj>	Corresponding adjectives, their synonyms and antonyms
<object>	Alphanumeric field names, synonyms, and key field values, optionally preceded by an article and/or a modifier and/or a superlative form of a corresponding adjective followed by the key-field or key-field synonym. Also defined recursively as: <object> {AND <object>}
<action>	WH? <object> | HOW MANY|MUCH <object> | <request> {THE <number> OF} <object> WHICH|WHO|THAT |
<number>	NUMBER | AMOUNT | QUANTITY
<boolean>	Boolean field names and synonyms
<numeric>	Numeric field names and synonyms
<pp>	Prepositional phrase, defined as: <prep> <numeric-field> {OF} <comparator> | <prep> <comparator> <numeric-field> |

<prep> {<art>} <alpha-value> <alpha-field-name> |
<prep> {<art>} <superlative> <matching-field-name> |
<prep> <boolean-field-name> |
<prep> <key-field-name> |
<pp> AND <pp>

<prep> Preposition (IN, AT, WITH, ON, etc.)

<art> Article (A, AN, THE, etc.)

<comparator> Comparative clause, defined as:
{comparative>} (no comparative implies equals)
(<number> | <numeric-field> | <key-value>)
{units-of-measure {corr-adj}}

<superlative> -Est form of corresponding adjectives

<comparative> {NOT|NO} -Er form of corresponding adjectives
THAN | {NO} GREATER THAN | {NO} LESS THAN |
{NOT} EQUAL TO | {NO} MORE THAN | AT LEAST

<request> One of the phrases "DISPLAY {ALL}",
"GIVE ME {ALL}", or "{TELL ME} WH? BE"

<verb-intrans> Corresponding intransitive verbs

<verb-trans> Corresponding transitive verbs

Examples of valid input are as follows:

How many years old are John Smith and Sally Jones?
How long are ships with age older than 15?
How wealthy is the oldest accounting department employee?
How tall are the married employees in the tax department?
How heavy is the lightest ship and the heaviest ship with cannons?
How tall is the shortest employee?
How many carriers have cannons and aircraft?
How many married tax department employees have pensions?
What ships exist?
How many employees work in the accounting department?
How many missiles and aircraft are carried by the CV-72?
What is the number of battleships with missiles and cannons?
Give me all ships of no greater than 1500 tons of weight.
Are there any ships?
Is John Smith married?
How much does Jim Brown weigh?
How much was collected by the IRS in 1979?

The ATN parser used is the SNePS ATN parser. The parser calls the following LISP functions:

```
(defun find-values (value-list)
   (cond
      ((null value-list) nil)
      (t (append (cdadar value-list) (find-values (cdr value-list))))))

(defun get-value (value-list data-value)
   (cond
      ((null value-list) nil)
      ((not (equal data-value (cdadar value-list)))
         (get-value (cdr value-list) data-value))
      (t (list (cdadar (find member (^(caar value-list)) file ?x))
               '|.|
               (caar value-list)
               '=
               (car data-value)))))

;valid digits expressed as atoms
(setq numbers '(\\ |0| |1| |2| |3| |4| |5| |6| |7| |8| |9|))

;determines if an atom is composed entirely of digits
(defun isnumber (lst)
   (cond
      ((null lst) t)
      ((member (car lst) numbers) (isnumber (cdr lst)))
      (t nil)))

;returns the value passed to it
(defun echo (echoname) echoname)

;reads the next value from the keyboard and converts it to an atom
(defun read-atom () (concat (read)))

;if an atom contains spaces, returns only the portion after the last space
(defun last-word (atm)
   (cond
      ((member '| | (explode atm))
         (last-word (implode (cdr (member '| | (explode atm))))))
      (t atm)))

;converts a list of atoms into one large atom name
(defun atomize (lst)
   (cond
      ((null lst) nil)
```

```
      ((atom lst) lst)
      ((null (cdr lst)) (car lst))
      (t (concat (car lst) '| | (atomize (cdr lst))))))

;atomizes list, then finds the root for a atom (if any) in the lexicon
(defun root-of (atm)
   (cond
      ((atom atm)
         (get-root (car (lookup atm))))
      ((equal (length atm) 1)
         (get-root (car (lookup (atomize atm)))))
      (t
         (atomize (append
            (reverse (cdr (reverse atm)))
            (get-root (car (lookup (car (reverse atm))))))))))

;extracts the root from the lexical description of the word
(defun get-root (lex-list)
   (cond
      ((null lex-list) nil)
      ((equal (caar lex-list) 'root) (cdar lex-list))
      (t (get-root (cdr lex-list)))))
```

The ATN source code consists of the following arcs:

```
(s

   ;start of sentence
   (wrd (how)       t
      (to s/how))

   ;action phrase
   (push action t
      (setr params *)
      (setr qualification (cadr (getr params)))
      (setr retrieval (car (getr params)))
      (to s/action))

   ;request-type phrase
   (push request t
      (to s/request))

   ;be...
   (cat ident t
      (to s/be))

   ;wh-...
```

```
   (cat wh- t
      (setr operation 'list)
      (to s/wh-)))

(s/how

   ;"How many|much..."
   (wrd (many much) t
      (to s/how/many))

   ;"How <adjective>..."
   (push max-adj t
      (setr retrieval *)
      (to s/how/adj))

   (push min-adj t
      (setr retrieval *)
      (to s/how/adj)))


(s/how/many

   ;"How many <units>..."
   (to (s/units)
      (member
         (root-of (addr unit-reg *))
         (find-values (find field- (find unit ?x))))
      (setr field-name (car (find-values
         (find unit (^(root-of (getr unit-reg))) field ?x))))
      (setr retrieval (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;"How much..."
   (jump s/wh- t
      (setr operation 'total)))


(s/units

   ;"How many <units> <max-adj>..."
   (push max-adj t  ;the 't' shouldn't be here, but I couldn't get it to work
      (equal (getr retrieval) (getr *))
      (to s/how/adj))

   ;"How many <units> of..."
   (wrd (of) t
      (to s/units/of))
```

```
   ;"How many <units>..."
   (jump s/how/adj t))


(s/units/of
   ;"How many <units> of <num-field>
   (push num-field t
      (to s/how/adj)))


(s/how/adj

   ;"How <adjective>|(many <units> {of <num-field>}) BE..."
   (cat ident      t
      (to s/how/be))

   ;"How many <units> {of <num-field>}
   (jump s/wh- t
      (setr operation 'total)))


(s/how/be

   ;"How <adjective> BE <object>..."
   (push object    t
      (setr params *)
      (setr qualification
         (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (setr operation 'list)
      (to s/end)))


(s/action

   ;<action> has|have
   (cat aux t
      (to s/have))

   ;<action> be
   (cat ident t
      (to s/have))

   ;<action> <pp>*
```

```
   (push pp t
      (setr qualification
         (cond
            ((and (getr qualification) (getr *))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (getr *) (getr qualification)))
      (to s/action/pp))

   (jump s/action/pp t))


(s/have

   ;<action phrase> have <boolean field>
   (push bool-field t
      (setr qualification
         (cond
            ((and (getr qualification) (getr *))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (getr *) (getr qualification)))
      (to s/end)))


(s/action/pp

   ;<action phrase> <pp>* <intrans-verb>
   (to (s/end)
      (setr filename
         (car (find-values
            (find file ?x intrans (^(getr *))))))))


(s/request

   ;preceding determiner
   (cat det t
      (to s/request))

   ;<request> {the} <number>
   (cat quantity t
      (to s/request/number))

   ;<request>
   (jump s/request/of t
      (setr operation 'list)))
```

```
(s/request/number

   ;<request> the <number> of
   (wrd (of) t
      (setr operation 'count)
      (to s/request/of)))


(s/request/of

   ;<request> {the <number> of} <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (to s/end)))


(s/be

   ;be there...
   (wrd (there) t
      (to s/be/there)))


(s/be/there

   ;optional "any"
   (wrd (any) t
      (to s/be/there))

   ;be there {any} <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (setr operation 'yesno)
      (to s/end)))
```

```
(s/wh-

   ;wh- {num-field}
   (push num-field t
      (setr retrieval *)
      (to s/wh-))

   ;wh- {alpha-field}
   (push alpha-field t
      (setr retrieval *)
      (to s/wh-))

   ;wh- be
   (cat ident t
      (to s/wh-/be))

   ;wh- do
   (to (s/wh-/be)
      (equal (root-of (getr *)) 'do)))


(s/wh-/be

   ;wh- be|do <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (to s/wh-/object))

   ;wh- be|do
   (jump s/wh-/object t))


(s/wh-/object

   ;wh- be|do {object} <trans-verb>
   (to (s/end)
      (setr fieldname (car
         (find-values (find trans (^(root-of (getr *))) field ?x))))
      (setr retrieval (list
         (cadadar (find member (^(getr fieldname)) file ?x))
         '|.|
```

```
        (getr fieldname)))))

(s/end

   ;optional trailing prepositional phrases
   (push pp t
      (setr qualification
          (cond
            ((and (getr qualification) (getr *))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (getr *) (getr qualification)))
      (to s/end))

   ;end of sentence
   (jump s/post t
      (setr filenames
         (get-filenames (append (getr retrieval) (getr qualification))))
      (setr join
         (cond
            ((null (cdr (getr filenames))) nil)
            ((setr fromfield
               (car (find-values (find from-file (^(car (getr filenames)))
                                         to-file (^(cadr (getr filenames)))
                                         from-field ?x))))
               (list
                  (car (getr filenames))
                  '|.|
                  (getr fromfield)
                  (cadr (getr filenames))
                  '|.|
                  (car (find-values (find from-file (^(car (getr filenames)))
                                            to-file (^(cadr (getr filenames)))
                                            to-field ?x)))))
            ((setr fromfield
               (car (find-values (find from-file (^(cadr (getr filenames)))
                                         to-file (^(car (getr filenames)))
                                         from-field ?x))))
               (list
                  (cadr (getr filenames))
                  '|.|
                  (getr fromfield)
                  (car (getr filenames))
                  '|.|
                  (car (find-values (find from-file (^(cadr (getr filenames)))
                                            to-file (^(car (getr filenames)))
                                            to-field ?x)))))
            (t '(error))))))
```

```
(s/post

   ;post-process registers
   (pop (post-process (getr retrieval) (getr qualification)
                      (getr operation) (getr join)) t))


(num-field

   ;numeric field name
   (to (num-field/end)
      (member
         (root-of (addr phrase *))
         (find-values (find type numeric member ?x)))
      (setr numfield (list
         (cadadar (find member (^(getr phrase)) file ?x))
         '|.|
         (getr phrase))))

   ;numeric field synonyms
   (to (num-field/end)
      (member
         (setr field-name (car (find-values
            (find synonym (^(root-of (addr phrase *))) field ?x))))
         (find-values (find type numeric member ?x)))
      (setr numfield (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;may be a phrase rather than a word
   (to (num-field) t
      (addr phrase *)))


(num-field/end

   ;numeric field or synonym found
   (pop numfield t))


(key-field

   ;key field name
   (to (key-field/end)
      (member
         (root-of (addr phrase *))
```

```
         (find-values (find file- (find key ?x))))
      (setr keyfield (list
         (cadadar (find key (^(getr phrase)) file ?x))
         '|.|
         (getr phrase))))

   ;key field synonyms
   (to (key-field/end)
      (member
         (setr field-name (car (find-values
            (find synonym (^(root-of (addr phrase *))) field ?x))))
         (find-values (find file- (find key ?x))))
      (setr keyfield (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;may be a phrase rather than a word
   (to (key-field) t
      (addr phrase *)))


(key-field/end

   ;key field or synonym found
   (pop keyfield t))


(alpha-field

   ;alphanumeric field name
   (to (alpha-field/end)
      (member
         (root-of (addr phrase *))
         (find-values (find type alphanumeric member ?x)))
      (setr alphafield (list
         (cadadar (find member (^(getr phrase)) file ?x))
         '|.|
         (getr phrase))))

   ;alphanumeric field synonyms
   (to (alpha-field/end)
      (member
         (setr field-name (car (find-values
            (find synonym (^(root-of (addr phrase *))) field ?x))))
         (find-values (find type alphanumeric member ?x)))
      (setr alphafield (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
```

```
         (getr field-name))))

   ;may be a phrase rather than a word
   (to (alpha-field) t
      (addr phrase *)))


(alpha-field/end

   ;alphanumeric field or synonym found
   (pop alphafield t))


(bool-field

   ;boolean field name
   (to (bool-field/end)
      (member
         (root-of (addr phrase *))
         (find-values (find type boolean member ?x)))
      (setr boolfield (list
         (cadadar (find member (^(getr phrase)) file ?x))
         '|.|
         (getr phrase)
         '=
         'true)))

   ;boolean field synonyms
   (to (bool-field/end)
      (member
         (setr field-name (car (find-values
            (find synonym (^(root-of (addr phrase *))) field ?x))))
         (find-values (find type boolean member ?x)))
      (setr boolfield (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name)
         '=
         'true)))

   ;may be a phrase rather than a word
   (to (bool-field) t
      (addr phrase *)))


(bool-field/end

   ;boolean field or synonym found
   (pop boolfield t))
```

```
(max-adj

   ;maximizing adjective
   (to (adj/end)
      (member (root-of (addr adj-name *)) (find-values
         (find field- (find max-adj ?x))))
      (setr field-name (car (find-values
         (find max-adj (^(root-of (getr adj-name))) field ?x))))
      (setr retrieval (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;may be a phrase rather than a word
   (to (max-adj) (addr adj-name *)))

(min-adj

   ;minimizing adjective
   (to (adj/end)
      (member (root-of (addr adj-name *)) (find-values
         (find field- (find min-adj ?x))))
      (setr field-name (car (find-values
         (find min-adj (^(root-of (getr adj-name))) field ?x))))
      (setr retrieval (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;may be a phrase rather than a word
   (to (min-adj) (addr adj-name *)))


(adj/end

   ;adjective found
   (pop  retrieval t))


(object

   ;object may be a field name
   (jump object/field t)

   ;object may be a phrase
   (jump object/phrase t))
```

```
(object/phrase

   ;preceding determiner
   (cat det        t
      (to object))

   ;data file names
   (to (object/end)
      (setr filename
         (car (member (root-of (addr object-name *)) (find-values
            (find class filenames member ?x)))))
      (setr retrieval (append (getr retrieval) (list
         (getr filename)
         '|.|
         (car (find-values (find file (^(getr filename)) key ?x)))))))

   ;file name synonyms
   (to (object/end)
      (setr filename
         (car (find-values
            (find synonym (^(root-of (addr object-name *))) file ?x))))
      (setr retrieval (append (getr retrieval) (list
         (getr filename)
         '|.|
         (car (find-values (find file (^(getr filename)) key ?x)))))))

   ;alphanumeric data field values
   (to (object/alpha-value)
      (setr field-name (car (find-values
         (find value (^(root-of (addr object-name *))) field ?x))))
      (setr qualification
          (cond
            ((getr qualification)
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name)
         '=
         (root-of (getr object-name)))
         (getr qualification)))
      (setr retrieval (append (getr retrieval) (list
        (cadadar (find member (^(getr field-name)) file ?x))
        '|.|
        (getr field-name)))))

   ;object may be a phrase rather than a word
```

```
   (to (object/phrase) (addr object-name *)))


(object/alpha-value

   ;<alpha-value> <alpha-field>
   (push alpha-field t
      (equal (getr field-name) (setr field-name-1 *))
      (to object/alpha-field))

   ;<alpha-value>
   (jump object/end t))


(object/alpha-field

   ;<alpha-value> <alpha-field> <data-file>
   (push object t
      (setr params *)
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (to object/end))

   ;<alpha-value> <alpha-field>
   (jump object/end t))


(object/field

   ;alphanumeric data field names & synonyms
   (push alpha-field t
      (setr retrieval (append (getr retrieval) (getr *)))
      (to object/end))

   ;boolean field names & synonyms
   (push bool-field t
      (setr qualification
         (cond
            ((and (getr qualification) (getr *))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (getr *) (getr retrieval)))
      (to object/est))

   ;superlative form of maximizing adjective
   (push max-adj t
      (setr qualification
         (cond
            ((and (getr qualification) (getr *))
               (append (getr qualification) '(and)))
```

```
            (t (getr qualification))))
      (setr qualification (append (getr qualification) '(greatest)))
      (setr qualification (append (getr qualification) (getr *)))
      (to object/est))

   ;superlative form of minimizing adjective
   (push min-adj t
      (setr qualification
         (cond
            ((and (getr qualification) (getr *))
               (append (getr qualification) '(and)))
            (t (getr qualification))))
      (setr qualification (append (getr qualification) '(least)))
      (setr qualification (append (getr qualification) (getr *)))
      (to object/est))

   ;key field names & synonyms
   (push key-field t
      (setr retrieval (append (getr retrieval) (getr *)))
      (to object/end)))

(object/est

   ;<superlative> <object>
   (push object t
      (setr params *)
      (setr qualification
         (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (to object/end)))


(object/end

   ;object found
   (pop (list (getr retrieval) (getr qualification)) t))


(action

   ;"what|which"
   (wrd (what which) t
      (to action/what))

   ;"how"
```

```
   (wrd (how) t
      (to action/how))

   ;request-type phrase
   (push request t
      (to action/request)))


(action/what

   ;wh? <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (liftr operation 'list)
      (to action/end)))


(action/how

   ;how many|much
   (wrd (many much) t
      (to action/how/many)))


(action/how/many

   ;how many|much <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (liftr operation 'count)
      (to action/end)))


(action/request
```

```
   ;preceding determiner
   (cat det t
      (to action/request))

   ;<request> {the} number|amount|quantity
   (cat quantity t
      (to action/amount))

   (jump action/of t
      (liftr operation 'list)))


(action/amount

   ;<request> the number of
   (wrd (of) t
      (liftr operation 'count)
      (to action/of)))


(action/of

   ;<request> the number of <object>
   (push object t
      (setr params *)
      (setr qualification
          (cond
            ((and (getr qualification) (cadr (getr params)))
               (cons 'and (getr qualification)))
            (t (getr qualification))))
      (setr qualification (append (cadr (getr params)) (getr qualification)))
      (setr retrieval (append (car (getr params)) (getr retrieval)))
      (to action/object)))


(action/object

   ;<request> the number of <object> which|who|that
   (cat relclause t
      (to action/end)))


(action/end

   ;end of action phrase
   (pop (list (getr retrieval) (getr qualification)) t))


(request
```

```
   ;"display"
   (wrd (display) t
      (to request/display))

   ;"give"
   (wrd (give) t
      (to request/give))

   ;"tell"
   (wrd (tell) t
      (to request/tell))

   (jump request/tell/me t))


(request/display

   ;"display all"
   (wrd (all) t
      (to request/end))

   ;"display"
   (jump request/end t))


(request/give

   ;"give me"
   (wrd (me) t
      (to request/give/me)))


(request/give/me

   ;"give me all"
   (wrd (all) t
      (to request/end))

   ;"give me"
   (jump request/end t))


(request/tell

   ;"tell me"
   (wrd (me) t
      (to request/tell/me)))
```

```
(request/tell/me

   ;{tell me} wh?
   (cat wh- t
      (to request/what)))


(request/what

   ;{tell me} wh? be
   (cat ident t
      (to request/end)))


(request/end

   ;request phrase found
   (pop nil t))


(pp

   ;preposition
   (cat prep       t
      (to pp/prep)))


(pp/prep

   ;numeric field or synonym
   (push num-field t
      (setr retrieval *)
      (to pp/num))

   ;boolean field or synonym
   (push bool-field t
      (setr retrieval *)
      (to pp/end))

   ;comparator
   (push comp t
      (setr retrieval *)
      (to pp/comp))

   ;try options which permit a preceding determiner
   (jump pp/det t))
```

```
(pp/comp

   ;<preposition> <comparator> <numeric>
   (push num-field t
      (setr retrieval (append (getr *) (getr retrieval)))
      (to pp/end)))


(pp/num

   ;preceding "of"
   (wrd (of)     t
      (to pp/num))

   ;"<preposition> <numeric-field> {of} <comparator>
   (push comp    t
      (setr retrieval (append (getr retrieval) (getr *)))
      (to pp/end)))


(pp/det

   ;preceding determiner
   (cat det t
      (to pp/det))

   ;alphanumeric data field values
   (to (pp/value)
      (setr field-name (car (find-values
         (find value (^(root-of (addr object-name *))) field ?x))))
      (setr retrieval (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name)
         '=
         (root-of (getr object-name)))))

   ;superlative form of maximizing adjective
   (to (pp/maxest)
      (member (root-of (addr object-name *)) (find-values
         (find field- (find max-adj ?x))))
      (setr field-name (car (find-values
         (find max-adj (^(root-of (getr object-name))) field ?x))))
      (setr est-name (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;superlative form of minimizing adjective
```

```
   (to (pp/minest)
      (member (root-of (addr object-name *)) (find-values
         (find field- (find min-adj ?x))))
      (setr field-name (car (find-values
         (find min-adj (^(root-of (getr object-name))) field ?x))))
      (setr est-name (list
         (cadadar (find member (^(getr field-name)) file ?x))
         '|.|
         (getr field-name))))

   ;may be a phrase rather than a single word
   (to (pp/det) t
      (addr object-name *)))


(pp/value

   ;<preposition> {art} <alpha-value> <alpha-field-name>
   (to (pp/end)
      (equal (getr field-name) (setr field-name-1 *)))

   ;<preposition> {art} <alpha-value> <synonym>
   (to (pp/end)
      (member
         (root-of (addr synonym-name *))
         (find-values
            (find field (^(root-of (getr field-name))) synonym ?x))))

   ;<preposition> {art} <alpha-value>
   (jump pp/end t))


(pp/maxest

   ;<preposition> {art} <superlative> <matching field>
   (push num-field t
      (equal (root-of (getr est-name)) (root-of (setr retrieval *)))
      (setr retrieval (append 'greatest (getr retrieval)))
      (to pp/end)))


(pp/minest

   ;<preposition> {art} <superlative> <matching field>
   (push num-field t
      (equal (root-of (getr est-name)) (root-of (setr retrieval *)))
      (setr retrieval (append 'least (getr retrieval)))
      (to pp/end)))
```

```
(pp/end

   ;end of prepositional phrase
   (pop retrieval t))


(comp

   ;optional comparative phrase
   (push comparator t
      (setr comp-phrase *)
      (to comp/phrase))

   ;otherwise, comparator is "equals"
   (jump comp/phrase t
      (setr comp-phrase '(=))))


(comp/phrase

   ;numeric field or value
   (push number t
      (setr comp-phrase (append (getr comp-phrase) (getr *)))
      (to comp/number)))


(comp/number

   ;<comparator> <number> <unit-of-measure>
   (to (comp/units)
      (member
         (root-of (addr unit-reg *))
         (find-values (find field- (find unit ?x)))))

   ; <units> may be a phrase
   (to (comp/number) t
      (addr unit-reg *))

   ; <comparator> <number>
   (jump comp/end t))


(comp/units

   ;<comparator> <number> <unit-of-measure> <max-adj>
   (push max-adj t
      (to comp/end))
```

```
   ;<comparator> <number> <unit-of-measure> of
   (wrd (of) t
      (to comp/units/of))

   ;<comparator> <number> <unit-of-measure>
   (jump comp/end t))


(comp/units/of

   ;<comparator> <number> <unit-of-measure> of <num-field>
   (push numfield t
      (to comp/end)))


(comp/end

   ;end of comparative phrase
   (pop comp-phrase t))


(number

   ;number
   (to (number/end)
      (isnumber (explode (setr numeric *)))
      (setr numeric (list (getr numeric))))

   ;numeric field or synonym
   (push num-field t
      (setr numeric *)
      (to number/end)))


(number/end
   (pop numeric t))


(comparator

   ;preceding "not" or "no"
   (cat neg t
      (setr negate (not (getr negate)))
      (to comparator))

   ;"greater" or "more"
   (wrd (greater more) t
      (to comparator/more))
```

```
   ;"less"
   (wrd (less) t
      (to comparator/less))

   ;"equal"
   (wrd (equal) t
      (to comparator/equal))

   ;"at"
   (wrd (at)    t
      (to comparator/at)))


(comparator/more

   ;"{no|not} greater than" or "{no|not} more than"
   (wrd (than) t
      (cond
         ((getr negate) (setr symbol '(<=)))
         (t (setr symbol '(>))))
      (to comparator/end)))


(comparator/less

   ;"{not|no} less than"
   (wrd (than) t
      (cond
         ((getr negate) (setr symbol '(>=)))
         (t (setr symbol '(<))))
      (to comparator/end)))


(comparator/equal
   ;"{not} equal to"
   (wrd (to) t
      (cond
         ((getr negate) (setr symbol '(<>)))
         (t (setr symbol '(=))))
      (to comparator/end)))


(comparator/at
   ;"{not} at least"
   (wrd (least) t
      (cond
         ((getr negate) (setr symbol '(<)))
         (t (setr symbol '(>=))))
      (to comparator/end)))
```

```
(comparator/end
   ;pop symbol
   (pop symbol t))
```

APPENDIX D: The Lexicon

The original lexicon is quite small, reflecting the fact that context-specific terms are not known until interview process. The following terms constitute the permanent part of the lexicon which was used in the demonstration program:

```
(a              ((ctgy . det)))
(an             ((ctgy . det)))
(the            ((ctgy . det)))
(of             ((ctgy . prep)))
(at             ((ctgy . prep)))
(for            ((ctgy . prep)))
(in             ((ctgy . prep)))
(with           ((ctgy . prep)))
(by             ((ctgy . prep)))
(be             ((ctgy . ident) (pres . is) (past . was) (pastp . been)))
(is             ((ctgy . ident) (root . be) (tense . pres)))
(are            ((ctgy . ident) (root . be) (tense . pres)))
(am             ((ctgy . ident) (root . be) (tense . pres)))
(was            ((ctgy . ident) (root . be) (tense . past)))
(been           ((ctgy . ident) (root . be) (tense . pastp)))
(do             ((ctgy . v) (pres . does) (past . did) (pastp . done)))
(does           ((ctgy . v) (root . do) (tense . pres)))
(did            ((ctgy . v) (root . do) (tense . past)))
(done           ((ctgy . v) (root . do) (tense . pastp)))
(number         ((ctgy . quantity)))
(amount         ((ctgy . quantity)))
(quantity       ((ctgy . quantity)))
(what           ((ctgy . wh-)))
(where          ((ctgy . wh-)))
(when           ((ctgy . wh-)))
(who            ((ctgy . wh-)) ((ctgy . relclause)))
(which          ((ctgy . relclause)))
(that           ((ctgy . relclause)))
(no             ((ctgy . neg)))
(not            ((ctgy . neg)))
```

APPENDIX E: Post-Processor Source Listing

The demonstration program includes a post-processor which converts the intermediate-language registers into a set of formal queries. This code is the only section which is specific to a particular query language.

```
;top level of post processor
(defun post-process (retrieval qualification operation join)
      (princ "Contents of registers:")
      (princ (ascii 10))
      (princ "     retrieval: ")
      (princ retrieval)
      (princ (ascii 10))
      (princ "     qualification: ")
      (princ qualification)
      (princ (ascii 10))
      (princ "     operation: ")
      (princ operation)
      (princ (ascii 10))
      (princ "     join: ")
      (princ join)
      (princ (ascii 10))
      (princ (ascii 10))
      (setq line-number 0)
      (use-statements
         (merge (get-filenames retrieval) (get-filenames qualification)))
      (qual-statements qualification)
      (retr-statements (get-fieldnames retrieval) operation join))

;generate USE statements
(defun use-statements (filename-list)
   (cond
      ((null filename-list) nil)
      (t
         (setq line-number (+ 1 line-number))
         (set (car filename-list) line-number)
         (princ line-number)
         (princ ". USE ")
         (princ (car filename-list))
         (princ (ascii 10))
         (use-statements (cdr filename-list)))))
```

```
;generate statements which qualify the chosen files
(defun qual-statements (qualification)
   (cond
      ((null qualification) nil)
      ((equal (car qualification) 'greatest)
         (princ (+ 1 line-number))
         (princ ". %GREATEST = 0")
         (princ (ascii 10))
         (princ (+ 2 line-number))
         (princ ". FOR ALL RECORDS IN ")
         (princ (eval (cadr qualification)))
         (princ (ascii 10))
         (princ "   IF ")
         (princ (nthelem 4 qualification))
         (princ " > %GREATEST THEN")
         (princ (ascii 10))
         (princ "      %GREATEST = ")
         (princ (nthelem 4 qualification))
         (princ (ascii 10))
         (setq line-number (+ 3 line-number))
         (princ line-number)
         (princ ". FIND ALL RECORDS IN ")
         (princ (eval (cadr qualification)))
         (set (cadr qualification) line-number)
         (princ " FOR WHICH ")
         (princ (nthelem 4 qualification))
         (princ " = %GREATEST")
         (princ (ascii 10))
         (qual-statements (cddddr qualification)))
      ((equal (car qualification) 'least)
         (princ (+ 1 line-number))
         (princ ". %LEAST = 0")
         (princ (ascii 10))
         (princ (+ 2 line-number))
         (princ ". FOR ALL RECORDS IN ")
         (princ (eval (cadr qualification)))
         (princ (ascii 10))
         (princ "   IF ")
         (princ (nthelem 4 qualification))
         (princ " < %LEAST THEN")
         (princ (ascii 10))
         (princ "      %LEAST = ")
         (princ (nthelem 4 qualification))
         (princ (ascii 10))
         (setq line-number (+ 3 line-number))
         (princ line-number)
         (princ ". FIND ALL RECORDS IN ")
         (princ (eval (cadr qualification)))
         (set (cadr qualification) line-number)
```

```
            (princ " FOR WHICH ")
            (princ (nthelem 4 qualification))
            (princ " = %LEAST")
            (princ (ascii 10))
            (qual-statements (cddddddr qualification)))
         (t
            (setq line-number (+ 1 line-number))
            (princ line-number)
            (princ ". FIND ALL RECORDS IN ")
            (princ (eval (car qualification)))
            (set (car qualification) line-number)
            (princ " FOR WHICH ")
            (princ (nthelem 3 qualification))
            (princ " ")
            (princ (nthelem 4 qualification))
            (princ " ")
            (princ (nthelem 5 qualification))
            (princ (ascii 10))
            (qual-statements (cddddddr qualification)))))

;generate retrieval statement
(defun retr-statements (fieldnames operation join)
   (cond
      ((equal operation 'list) (list-statement fieldnames join))
      ((equal operation 'count) (count-statement join))
      ((equal operation 'total) (total-statement fieldnames join))
      ((equal operation 'yesno) (yesno-statement join))
      (t
         (princ "Illegal value in operation register")
         (princ (ascii 10)))))

;generate retrieval statement if operation is list
(defun list-statement (fieldnames join)
   (cond
      ((null fieldnames) nil)
      (t
         (princ (+ 1 line-number))
         (cond
            ((null join)
               (princ ". FOR EACH RECORD IN ")
               (princ line-number))
            (t
               (princ ". FOR EACH RECORD IN ")
               (princ (eval (nthelem 4 join)))
               (princ (ascii 10))
               (princ "     FOR EACH RECORD IN ")
               (princ (eval (car join)))
               (princ " FOR WHICH ")
               (princ (nthelem 6 join))
```

```
            (princ " = ")
            (princ (nthelem 3 join))))
        (princ (ascii 10))
        (princ "      PRINT ")
        (princ (car fieldnames))
        (additional-retr (cdr fieldnames)))))

;generate the rest of the retrieval statement if operation is list
(defun additional-retr (fieldnames)
   (cond
      ((null fieldnames) (princ (ascii 10)))
      (t
         (princ " AND ")
         (princ (car fieldnames))
         (additional-retr (cdr fieldnames)))))

;generate retrieval statement if operation is count
(defun count-statement (join)
   (princ (+ 1 line-number))
   (cond
      ((null join)
         (princ ". PRINT COUNT IN ")
         (princ line-number)
         (princ (ascii 10)))
      (t
         (princ ". FOR EACH RECORD IN ")
         (princ (eval (car join)))
         (princ (ascii 10))
         (princ "     PRINT COUNT IN ")
         (princ (eval (nthelem 4 join)))
         (princ " FOR WHICH ")
         (princ (nthelem 6 join))
         (princ " = ")
         (princ (nthelem 3 join)))))

;generate retrieval statement if operation is total
(defun total-statement (fieldnames join)
   (princ (+ 1 line-number))
   (princ ". %TOTAL = 0")
   (princ (ascii 10))
   (princ (+ 2 line-number))
   (cond
      ((null join)
         (princ ". FOR EACH RECORD IN ")
         (princ line-number)
         (princ (ascii 10))
         (princ "   %TOTAL = %TOTAL"))
      (t
         (princ ". FOR EACH RECORD IN ")
```

```
        (princ (eval (nthelem 4 join)))
        (princ (ascii 10))
        (princ "     FOR EACH RECORD IN ")
        (princ (eval (car join)))
        (princ " FOR WHICH ")
        (princ (nthelem 6 join))
        (princ " = ")
        (princ (nthelem 3 join))
        (princ (ascii 10))
        (princ "      %TOTAL = %TOTAL")))
   (additional-total fieldnames))

;generate rest of retrieval statement if operation is total
(defun additional-total (fieldnames)
   (cond
      ((null fieldnames)
         (princ (ascii 10))
         (princ (+ 3 line-number))
         (princ ". PRINT %TOTAL")
         (princ (ascii 10)))
      (t
         (princ " + ")
         (princ (car fieldnames))
         (additional-total (cdr fieldnames)))))

;generate retrieval statement is operation is yesno
(defun yesno-statement (join)
   (princ (+ 1 line-number))
   (cond
      ((null join) t)
      (t
         (princ ". FOR EACH RECORD IN ")
         (princ (eval (nthelem 4 join)))
         (princ (ascii 10))
         (princ "     FIND ALL RECORDS IN ")
         (princ (eval (car join)))
         (princ " FOR WHICH ")
         (princ (nthelem 6 join))
         (princ " = ")
         (princ (nthelem 3 join))
         (princ (ascii 10))
         (princ (+ 2 line-number))
         (setq line-number (+ 1 line-number))))
   (princ ". IF COUNT IN ")
   (princ line-number)
   (princ " = 0 THEN")
   (princ (ascii 10))
   (princ "      PRINT 'NO'")
   (princ (ascii 10))
```

```
   (princ "   ELSE")
   (princ (ascii 10))
   (princ "      PRINT 'YES'")
   (princ (ascii 10))
   (princ "   ENDIF")
   (princ (ascii 10)))

;merges two lists together, removing the duplicates
(defun merge (lst1 lst2)
   (cond
      ((null lst1) lst2)
      ((null lst2) lst1)
      ((member (car lst1) lst2) (merge (cdr lst1) lst2))
      (t (merge (cdr lst1) (cons (car lst1) lst2)))))

;gets a list of filenames stored in a register
(defun get-filenames (register)
   (cond
      ((null register) nil)
      ((null (cdr register)) nil)
      ((equal (cadr register) '|.|)
         (merge (list (car register)) (get-filenames (cdr register))))
      (t (get-filenames (cdr register)))))

;gets a list of fieldnames stored in a register
(defun get-fieldnames (register)
   (cond
      ((null register) nil)
      ((null (cdr register)) nil)
      ((equal (car register) '|.|)
         (merge (list (cadr register)) (get-fieldnames (cdr register))))
      (t (get-fieldnames (cdr register)))))

;gets a list of filenames stored in a register
(defun get-filenames (register)
   (cond
      ((null register) nil)
      ((null (cdr register)) nil)
      ((equal (cadr register) '|.|)
         (merge (list (car register)) (get-filenames (cdr register))))
      (t (get-filenames (cdr register)))))
```

APPENDIX F: Sample System Output

The following shows a sample transcript of a session with the example natural language understanding program developed in conjunction with this paper. The system first interviews the user as to the structure of the target data bases. It then accepts a series of queries and displays the intermediate-language registers and the formal query produced.

The example domain consists of two files: The first is called PERSONNEL-FILE and consists of employee information. Synonyms for the file name are EMPLOYEE and WORKER. Valid corresponding verbs are EXIST and WORK. The file consists of the following fields:

EMPLOYEE-NAME is the key value of the field. Its synonym is NAME. It is an alphanumeric field which modifies the data base file. Since it is an alphanumeric field, the system must know what data values are contained in the field. Valid data values include "John Doe", "James Smith", and "Bruce Springsteen".

EMPLOYEE-AGE is a numeric field which modifies the data base file. Its synonym is AGE. Its unit of measure is YEAR. It has maximizing corresponding adjectives OLD, AGED, and the frozen phrase "OVER THE HILL". It has a

minimizing corresponding adjective YOUNG. EMPLOYEE-SALARY is a numeric field which modifies the data base file. Its synonyms include SALARY and WAGE. Its unit of measure is DOLLAR. It has maximizing corresponding adjectives RICH and WEALTHY, and a minimizing corresponding adjective POOR. It has a corresponding verb EARN.

EMPLOYEE-DEPARTMENT is an alphanumeric field which modifies the data base file. Its synonym is DEPARTMENT. Sample data includes "tax", "accounting", and "data processing".

IS-MARRIED is a boolean field which modifies the data base file. Its synonym is MARRIED, and it has a corresponding verb MARRY.

The second data base file is called INSURANCE-RATES. The file consists of a list of insurance rates given an employee's age. The file includes the following fields:

INSURED-AGE is the key field. It is a numeric field. Its synonym is "insured age".

INSURANCE-COST is a numeric field. Its synonym is "whole life cost". It has a corresponding verb "cost", as in "How much does an insured person cost (the company)? It

also has a maximizing corresponding adjective EXPENSIVE, and minimizing corresponding adjectives INEXPENSIVE and CHEAP.

Since EMPLOYEE-AGE refers to the same thing as INSURED-AGE, a link exists between the two data base files. The link connects INSURANCE-RATES.INSURED-AGE to EMPLOYEE-FILE.EMPLOYEE-AGE. The link allows the system to answer queries concerning the insurance rates of individual employees.

Output from the program follows. Input from the user is underlined for clarity.

```
Script started on Mon Jun  8 10:37:29 1987
= project <2:1> => sneps
Franz Lisp, Opus 38.79

-------------------------
  Loading UB lisp changes
-------------------------

Mon Jun  8 10:37:55 1987

-------------------------
   lisprc.l init-loaded.
-------------------------

sneps
*(^(load "load1"))
[load load1]
[load englex.l]
[load functions.l]
[load getnet.l]
```

```
(member member-)
(class class-)
(synonym synonym-)
(file file-)
(key key-)
(max-adj max-adj-)
(min-adj min-adj-)
(type type-)
(field field-)
(value value-)
(unit unit-)
(trans trans-)
(intrans intrans-)
(from-file from-file-)
(from-field from-field-)
(to-file to-file-)
(to-field to-field-)
[load post.l]
Enter new database filename or end: personnel-file
Done!  Enter synonym for personnel-file or end: employee
Done!  Enter synonym for personnel-file or end: worker
Done!  Enter synonym for personnel-file or end: end
Enter corresponding verb for personnel-file or end: exist
Done!  Enter corresponding verb for personnel-file or end: work
Done!  Enter corresponding verb for personnel-file or end: end
Enter field name or end: employee-name
Done!  Enter synonym for employee-name or end: name
Done!  Enter synonym for employee-name or end: end
Is this field a key field? y
Done!  Enter corresponding verb for employee-name or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? a
Done!  Does this field modify the data base file? (Y/N) y
Done!  Enter data value for field or end: "John Doe"s
Done!  Enter data value for field or end: "James Smith"
Done!  Enter data value for field or end: "Bruce Springsteen"
Done!  Enter data value for field or end: end
Enter field name or end: employee-age
Done!  Enter synonym for employee-age or end: age
Done!  Enter synonym for employee-age or end: end
Is this field a key field? n
Enter corresponding verb for employee-age or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? n
Done!  Enter the name a proper unit of measure or end: year
Done!  Enter the name a proper unit of measure or end: end
Enter the name of a maximizing adjective or end: old
Done!  Enter the name of a maximizing adjective or end: aged
Done!  Enter the name of a maximizing adjective or end: "over the hill"
Done!  Enter the name of a maximizing adjective or end: end
Enter the name of a minimizing adjective or end: young
```

```
Done!  Enter the name of a minimizing adjective or end: end
Enter field name or end: employee-salary
Done!  Enter synonym for employee-salary or end: salary
Done!  Enter synonym for employee-salary or end: wage
Done!  Enter synonym for employee-salary or end: end
Is this field a key field? n
Enter corresponding verb for employee-salary or end: earn
Done!  Enter corresponding verb for employee-salary or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? n
Done!  Enter the name a proper unit of measure or end: dollar
Done!  Enter the name a proper unit of measure or end: end
Enter the name of a maximizing adjective or end: rich
Done!  Enter the name of a maximizing adjective or end: wealthy
Done!  Enter the name of a maximizing adjective or end: end
Enter the name of a minimizing adjective or end: poor
Done!  Enter the name of a minimizing adjective or end: end
Enter field name or end: employee-department
Done!  Enter synonym for employee-department or end: department
Done!  Enter synonym for employee-department or end: end
Is this field a key field? n
Enter corresponding verb for employee-department or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? a
Done!  Does this field modify the data base file? (Y/N) y
Done!  Enter data value for field or end: tax
Done!  Enter data value for field or end: accounting
Done!  Enter data value for field or end: "data processing"
Done!  Enter data value for field or end: end
Enter field name or end: is-married
Done!  Enter synonym for is-married or end: married
Done!  Enter synonym for is-married or end: end
Is this field a key field? n
Enter corresponding verb for is-married or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? b
Done!  Does this field modify the data base file? (Y/N) y Done!  Enter field name or end:
Enter new database filename or end: insurance-rates
Done!  Enter synonym for insurance-rates or end: "insurance rate"
Done!  Enter synonym for insurance-rates or end: end
Enter corresponding verb for insurance-rates or end: end
Enter field name or end: insured-age
Done!  Enter synonym for insured-age or end: end
Is this field a key field? y
Done!  Enter corresponding verb for insured-age or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? n
Done!  Enter the name a proper unit of measure or end: end
Enter the name of a maximizing adjective or end: end
Enter the name of a minimizing adjective or end: end
Enter field name or end: insurance-cost
Done!  Enter synonym for insurance-cost or end: "whole life cost"
Done!  Enter synonym for insurance-cost or end: cost
```

```
Done!  Enter synonym for insurance-cost or end: end
Is this field a key field? n
Enter corresponding verb for insurance-cost or end: cost
Done!  Enter corresponding verb for insurance-cost or end: end
Is field (A)lpha, (N)umeric, or (B)oolean? n
Done!  Enter the name a proper unit of measure or end: dollar
Done!  Enter the name a proper unit of measure or end: end
Enter the name of a maximizing adjective or end: expensive
Done!  Enter the name of a maximizing adjective or end: end
Enter the name of a minimizing adjective or end: inexpensive
Done!  Enter the name of a minimizing adjective or end: cheap
Done!  Enter the name of a minimizing adjective or end: end
Enter field name or end: end
Enter new database filename or end: end
Enter origin file of link or end: insurance-rates
Enter origin field of link: insured-age
Enter destination file of link: personnel-file
Enter destination field of link: employee-age
Done! Enter origin file of link or end: end
(t)
exec: 6.83 sec    gc: 1.56 sec

*(^(parse))

atn parser initialization

(trace level= => 0 <=)

(beginning at state => s <=)

 Input sentences in normal English orthographic convention.
 May go beyond a line by having a space followed by <CR>.
 To end parser, write  ^end .


: ^(setq parse-trees t)

t

: how old is John Doe
Contents of registers:
     retrieval: (personnel-file . employee-name personnel-file . employee-age)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
```

```
     PRINT employee-name AND employee-age

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 2.200 gc= 1.633) <=)

: how over the hill are the married employees
Contents of registers:
     retrieval: (personnel-file . employee-name personnel-file . employee-age)
     qualification: (personnel-file . is-married = true)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH is-married = true
3. FOR EACH RECORD IN 2
     PRINT employee-name AND employee-age

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 4.533 gc= 0.00) <=)

: display the number of married workers
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . is-married = true)
     operation: count
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH is-married = true
3. PRINT COUNT IN 2

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 16.383 gc= 5.133) <=)

: tell me what is the number of employees who are married
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . is-married = true)
     operation: count
     join: nil
```

```
1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH is-married = true
3. PRINT COUNT IN 2

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 3.750 gc= 0.00) <=)

: give me all married employees
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . is-married = true)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH is-married = true
3. FOR EACH RECORD IN 2
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 7.900 gc= 3.300) <=)

: display all employees with whole life costs of at least 100 dollars
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost >= 100)
     operation: list
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE personnel-file
2. USE insurance-rates
3. FIND ALL RECORDS IN 2 FOR WHICH insurance-cost >= 100
4. FOR EACH RECORD IN 1
     FOR EACH RECORD IN 3 FOR WHICH employee-age = insured-age
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 28.816 gc= 8.333) <=)
```

```
: display the number of employees with whole life costs of at least 100 dollars
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost >= 100)
     operation: count
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE personnel-file
2. USE insurance-rates
3. FIND ALL RECORDS IN 2 FOR WHICH insurance-cost >= 100
4. FOR EACH RECORD IN 3
     PRINT COUNT IN 1 FOR WHICH employee-age = insured-age
valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 55.116 gc= 15.350) <=)

: are there any employees with whole life costs of at least 100 dollars
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost >= 100)
     operation: yesno
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE personnel-file
2. USE insurance-rates
3. FIND ALL RECORDS IN 2 FOR WHICH insurance-cost >= 100
4. FOR EACH RECORD IN 1
     FIND ALL RECORDS IN 3 FOR WHICH employee-age = insured-age
5. IF COUNT IN 4 = 0 THEN
     PRINT 'NO'
   ELSE
     PRINT 'YES'
   ENDIF

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 9.466 gc= 3.300) <=)

: how expensive is the oldest married employee in the data processing department
Contents of registers:
     retrieval: (personnel-file . employee-name
                 insurance-rates . insurance-cost)
     qualification: (personnel-file . employee-department = data processing and
                     personnel-file . is-married = true and
                     greatest personnel-file . employee-age)
```

```
     operation: list
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE insurance-rates
2. USE personnel-file
3. FIND ALL RECORDS IN 2 FOR WHICH employee-department = data processing
4. FIND ALL RECORDS IN 3 FOR WHICH is-married = true
5. %GREATEST = 0
6. FOR ALL RECORDS IN 4
   IF employee-age > %GREATEST THEN
      %GREATEST = employee-age
7. FIND ALL RECORDS IN 4 FOR WHICH employee-age = %GREATEST
8. FOR EACH RECORD IN 7
     FOR EACH RECORD IN 1 FOR WHICH employee-age = insured-age
      PRINT employee-name AND insurance-cost

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 18.566 gc= 3.500) <=)

: what employees are married with whole life costs of at least 100 dollars
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost >= 100 and
                     personnel-file . is-married = true)
     operation: list
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE insurance-rates
2. USE personnel-file
3. FIND ALL RECORDS IN 1 FOR WHICH insurance-cost >= 100
4. FIND ALL RECORDS IN 2 FOR WHICH is-married = true
5. FOR EACH RECORD IN 4
     FOR EACH RECORD IN 3 FOR WHICH employee-age = insured-age
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 11.466 gc= 3.383) <=)

: display all employees who exist
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: nil
     operation: list
```

```
     join: nil

1. USE personnel-file
2. FOR EACH RECORD IN 1
     PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 2.833 gc= 1.683) <=)

: give me the number of employees who work in the data processing department
with not more than 150 dollars of whole life costs
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost <= 150 and
                     personnel-file . employee-department = data processing)
     operation: count
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE insurance-rates
2. USE personnel-file
3. FIND ALL RECORDS IN 1 FOR WHICH insurance-cost <= 150
4. FIND ALL RECORDS IN 2 FOR WHICH employee-department = data processing
5. FOR EACH RECORD IN 3
     PRINT COUNT IN 4 FOR WHICH employee-age = insured-age
valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 27.600 gc= 6.816) <=)

: what married data processing department employees exist
Contents of registers:
     retrieval: (personnel-file . employee-name
                 personnel-file . employee-department)
     qualification: (personnel-file . employee-department = data processing and
                     personnel-file . is-married = true)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-department = data processing
3. FIND ALL RECORDS IN 2 FOR WHICH is-married = true
4. FOR EACH RECORD IN 3
     PRINT employee-name AND employee-department

valid parse or structure>
```

```
t
end of parse

(time (sec.) : => (cpu= 7.900 gc= 1.750) <=)

: display all employees
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: nil
     operation: list
     join: nil

1. USE personnel-file
2. FOR EACH RECORD IN 1
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 4.583 gc= 1.683) <=)

: display the oldest employee
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (greatest personnel-file . employee-age)
     operation: list
     join: nil

1. USE personnel-file
2. %GREATEST = 0
3. FOR ALL RECORDS IN 1
   IF employee-age > %GREATEST THEN
      %GREATEST = employee-age
4. FIND ALL RECORDS IN 1 FOR WHICH employee-age = %GREATEST
5. FOR EACH RECORD IN 4
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 11.00 gc= 3.333) <=)

: tell me what is the number of employees with whole life costs of greater
than 500 dollars
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost > 500)
```

```
     operation: count
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE personnel-file
2. USE insurance-rates
3. FIND ALL RECORDS IN 2 FOR WHICH insurance-cost > 500
4. FOR EACH RECORD IN 3
     PRINT COUNT IN 1 FOR WHICH employee-age = insured-age
valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 55.116 gc= 14.833) <=)

: are there any married employees in the tax department
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . employee-department = tax and
                     personnel-file . is-married = true)
     operation: yesno
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-department = tax
3. FIND ALL RECORDS IN 2 FOR WHICH is-married = true
4. IF COUNT IN 3 = 0 THEN
     PRINT 'NO'
   ELSE
     PRINT 'YES'
   ENDIF

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 6.633 gc= 1.683) <=)

: is there a worker in the tax department with at least 50 dollars of whole
life costs
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (insurance-rates . insurance-cost >= 50 and
                     personnel-file . employee-department = tax)
     operation: yesno
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE insurance-rates
2. USE personnel-file
3. FIND ALL RECORDS IN 1 FOR WHICH insurance-cost >= 50
```

```
4. FIND ALL RECORDS IN 2 FOR WHICH employee-department = tax
5. FOR EACH RECORD IN 4
     FIND ALL RECORDS IN 3 FOR WHICH employee-age = insured-age
6. IF COUNT IN 5 = 0 THEN
      PRINT 'NO'
   ELSE
      PRINT 'YES'
   ENDIF

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 33.300 gc= 10.366) <=)

: is there a John Doe in data processing
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . employee-department = data processing and
                     personnel-file . employee-name = John Doe)
     operation: yesno
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-department = data processing
3. FIND ALL RECORDS IN 2 FOR WHICH employee-name = John Doe
4. IF COUNT IN 3 = 0 THEN
      PRINT 'NO'
   ELSE
      PRINT 'YES'
   ENDIF

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 10.266 gc= 1.683) <=)

: how many years old is John Doe
Contents of registers:
     retrieval: (personnel-file . employee-name personnel-file . employee-age)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
      PRINT employee-name AND employee-age
```

```
valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 2.300 gc= 1.700) <=)

: how many years of age is John Doe
Contents of registers:
     retrieval: (personnel-file . employee-name personnel-file . employee-age)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
      PRINT employee-name AND employee-age

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 3.083 gc= 0.00) <=)

: who is John Doe
Contents of registers:
     retrieval: (personnel-file . employee-name)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
      PRINT employee-name

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 3.800 gc= 1.666) <=)

: who is the youngest married data processing department employee
Contents of registers:
     retrieval: (personnel-file . employee-name
                 personnel-file . employee-department)
     qualification: (personnel-file . employee-department = data processing and
                     personnel-file . is-married = true and
```

```
                    least personnel-file . employee-age)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-department = data processing
3. FIND ALL RECORDS IN 2 FOR WHICH is-married = true
4. %LEAST = 0
5. FOR ALL RECORDS IN 3
   IF employee-age < %LEAST THEN
      %LEAST = employee-age
6. FIND ALL RECORDS IN 3 FOR WHICH employee-age = %LEAST
7. FOR EACH RECORD IN 6
      PRINT employee-name AND employee-department

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 55.583 gc= 13.716) <=)

: what does John Doe earn
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
      PRINT employee-salary

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 6.833 gc= 1.683) <=)

: how much does John Doe cost
Contents of registers:
     retrieval: (insurance-rates . insurance-cost)
     qualification: (personnel-file . employee-name = John Doe)
     operation: total
     join: (insurance-rates . insured-age personnel-file . employee-age)

1. USE insurance-rates
2. USE personnel-file
3. FIND ALL RECORDS IN 2 FOR WHICH employee-name = John Doe
```

```
4. %TOTAL = 0
5. FOR EACH RECORD IN 3
     FOR EACH RECORD IN 1 FOR WHICH employee-age = insured-age
      %TOTAL = %TOTAL + insurance-cost
6. PRINT %TOTAL

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 3.800 gc= 1.700) <=)

: what is earned by John Doe
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
     PRINT employee-salary

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 16.400 gc= 5.100) <=)

: what is John Doe earning
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
     PRINT employee-salary

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 12.983 gc= 3.416) <=)

: what salary does John Doe earn
```

```
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: list
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. FOR EACH RECORD IN 2
     PRINT employee-salary

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 8.116 gc= 1.683) <=)

: how much does John Doe earn
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: total
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. %TOTAL = 0
4. FOR EACH RECORD IN 2
   %TOTAL = %TOTAL + employee-salary
5. PRINT %TOTAL

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 3.750 gc= 1.733) <=)

: how many dollars does John Doe earn
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: total
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. %TOTAL = 0
4. FOR EACH RECORD IN 2
   %TOTAL = %TOTAL + employee-salary
```

```
5. PRINT %TOTAL

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 4.450 gc= 0.00) <=)

: how many dollars of salary does John Doe earn
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-name = John Doe)
     operation: total
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-name = John Doe
3. %TOTAL = 0
4. FOR EACH RECORD IN 2
   %TOTAL = %TOTAL + employee-salary
5. PRINT %TOTAL

valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 4.900 gc= 1.700) <=)

: how much does the oldest married data processing department employee earn
Contents of registers:
     retrieval: (personnel-file . employee-salary)
     qualification: (personnel-file . employee-department = data processing and
                     personnel-file . is-married = true and
                     greatest personnel-file . employee-age)
     operation: total
     join: nil

1. USE personnel-file
2. FIND ALL RECORDS IN 1 FOR WHICH employee-department = data processing
3. FIND ALL RECORDS IN 2 FOR WHICH is-married = true
4. %GREATEST = 0
5. FOR ALL RECORDS IN 3
   IF employee-age > %GREATEST THEN
      %GREATEST = employee-age
6. FIND ALL RECORDS IN 3 FOR WHICH employee-age = %GREATEST
7. %TOTAL = 0
8. FOR EACH RECORD IN 6
   %TOTAL = %TOTAL + employee-salary
9. PRINT %TOTAL
```

```
valid parse or structure>
t
end of parse

(time (sec.) : => (cpu= 19.350 gc= 5.150) <=)

: ^end
(end atn parser)
exec: 418.61 sec     gc: 113.75 sec

*(exit)
No files updated.
== project <2:2> ==> ^D
script done on Mon Jun  8 11:11:44 1987
```

The vita has been removed from the scanned document