1985

Using natural language for database queries

Mikel J. Brown

Using Natural Language for

Database Queries

by

Mikel J. Brown

This thesis is submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science at Rochester Institute of Technology.

Approved by:

Thesis Advisor: Professor John A. Biles — 6/14/85 date

Graduate Director: Professor Peter G. Anderson — 14 June 1985 date

Committee Member: Professor Lawrence A. Coon — 6/17/85 date

Dedicated to our little Princess

Sarah Beth

Using Natural Language for
Database Queries

Natural Language Processing is described, together with a small natural language front-end processor called NBASE. The requirements and goals of a natural language system along with existing alternatives, are discussed. The necessity of preprocessing a knowledge base and the implementation of JPREP, a preprocessing interface to NBASE, is presented. Lexical analysis, natural language parsing, and semantic processing of natural language inputs, along with heuristics for transformation at each stage are offered. The interface to a formal query database (Mistress) is given. Difficulties of natural language implementations are exposed. A very general overview of PROLOG and its strength as a logic oriented language for natural language processing is covered, along with some ideas for further research in Natural language processing and database access.

ACKNOWLEDGEMENTS

I would like to thank my thesis committee for their involvement in this project, particularly Mr. John A. Biles (Al) for his encouragement and time spent listening to ideas, returning suggestions, and editing the manuscript.

Much appreciation is given to my parents, who encouraged this undertaking and were a constant support emotionally and financially when the "chips were down."

I could never thank my wife, Julie, enough! Always supportive, and often the receiving block of my frustrations experienced during this project. Always ready to help with the manuscript, thanks for your love! I know that I have found a "virtuous woman" (Proverbs 31:10-31)!

Most of all, I am eternally grateful to the LORD Jesus Christ, for His all sufficient grace, who in this thesis has assured me that man is "fearfully and wonderfully made."

PREFACE

This thesis examines the basic elements of a natural language processing system, while describing the implementation of a simple front end processor called NBASE. The major design issues will be addressed by referring in detail to existing alternatives implemented in several current systems (LUNAR, REL, INTELLECT and the LDC-1). The approach taken is to isolate the design issues by examining the strategies of these systems, commenting on their applicability to the growth of natural language processing and functionally relating them to the NBASE project. The method of describing the project is to describe system components which requires presenting theoretical concerns of the targeted systems, alternating with the functional specifications of NBASE. This approach is taken to enhance later efforts to upgrade this project or help instruction of particular components of natural language processing.

Finally, an alternative database approach is presented, describing a database system whose underlying structure and schema are defined by natural language constructs, as well as the access support to the database. The discussion in chapter 7 is not meant to be exhaustive, and the reader interested in further material on complete natural language Database Management Systems is referred to the bibliographic reference to Li (1984).

CONTENTS

1. INTRODUCTION

Intelligence is a concept that many have attempted to define, understand, measure, and, recently, emulate via machine (artificial intelligence or AI). It is debatable whether success has been achieved in any of these attempts to interpret intelligence, but some insights have emerged, particularly in the field of natural language processing.

Natural language analysis examines a machines' ability to "intelligently" respond to discourse[1], using a natural language such as English, German, French, etc. Turing's test for machine intelligence is a computers' ability to disguise itself as human while communicating with a human counterpart (Wilcox, 1983). Early successes in passing the Turing test include ELIZA (Weizenbaum, 1965), LUNAR (Woods, 1970), and SHRDLU (Winograd, 1972). While these projects established a foundation for natural language processing, their practical applications were severely limited.

The emergence of the "information explosion" has helped to make natural language processing and expert systems the two most promising and desired applications of AI research (Guida, 1983). The demand for a system that can process requests for information from users who have had little or no experience with the formalities of a computer system, has become a leading motivation for natural language processing. By exploiting a user's knowledge of his own natural language, a system can be produced that intelligently responds to just about any user's request. This was shown during the 1970's when several experimental natural language systems were developed, most of which directed at database queries (Bal-

[1] Discourse may occur using any one of many input methods. It includes but is not limited to audio communication and/or artificial methods such as via a keyboard.

lard & Lusth, 1982). Some of these products, such as the INTELLECT system developed by Harris (1984), have been expanded and marketed with varying degrees of success,

1.1. Why Natural Language ?

Accessing information in a database via query languages can be divided into four categories (Li, 1984):

1. Menu driven
2. Formal query language
3. Embedded language within a programming language (e.g. COBOL)
4. Natural language queries

Fixed script dialogues, or menus, represent the easiest query for an end user to operate. No experience is necessary, outside of knowing which tool, such as keys on a keyboard or a mouse, is the selector tool and which tool registers the users choice. Successive menu displays lead a user through the system until the desired information is located. The primary limitation of menu driven queries is that only queries specifically accommodated by the menus can be generated. This either limits the scope of information that can be accessed, or for very large information bases, potentially leads the user down many dead end paths irrelevant to his search (Leigh, 1983).

Formal and embedded languages increase the accessibility of the information base by isolating the table or other structure in which the desired data are likely to be found. However, several requirements force the user to become familiar with both the implemented query language and the structure of the knowledge base being queried.

One limiting requirement is that the syntax of formal languages is strict, requiring that the user express himself within often unnatural syntactic restrictions. The definitions of the language must be learned, and the organization of the language components examined in order to guarantee syntactically correct query formulations.

Furthermore, expressing queries in a formal query language requires specific knowledge of how the data base is structured (Harris, 1977b). The necessary information is defined by

a database schema, which includes, but is not limited to, specific relation and attribute names, and the structure of attribute values. Ideally, knowledge of these details should not be required of the end user.

Another consideration is that an experienced user's conceptualization and the actual structure of the database are different. This concept, to service the needs of all users, is a foundation of database theory, but it compounds the difficulties of inexperienced operators, particularly those who have little or no motivation for learning the query system (Date, 1983).

Like the other three methods, natural language query systems have their disadvantages. The major problems, however, are designing and implementing the query system, rather than in training the user. These disadvantages include the complexity of the semantic and syntactic structures and inherent ambiguities common to natural language. Furthermore, a general lack of linguistic theory to present an adequate, formalized theoretical base upon which to implement a system, requires that several heuristics be examined to provide an efficient knowledge retrieval system (Rhamstorf, 1979).

Providing simple access through natural language reduces a system's time efficiency for retrieving the desired data. The amount of processing required to analyze and derive meaning from a sentence requires increased processing time when compared to a formal system. For those users who are constrained to use a natural language processor, the longer response time is not perceived.

The chief advantage of a natural language query system is that there is little or no special training required to use it. There is no need to learn an artificial (formal) language, and knowledge of database schema is not required. A natural language system eliminates the necessity for the user to translate his natural language idea into a formal language statement, resulting in fewer mistakes and less time wasted. A natural language system is convenient; it facilitates efficiency of expression, yielding a versatile system. A natural language system also allows a user to start work while he is still uncertain how to express his request.

Natural language systems make it easy to phrase complicated questions, producing more powerful queries than menus while yielding simpler user interfaces than formal language systems. Indeed, many systems provide heuristics for context dependency so that incomplete queries are resolved using inferences developed from previous queries (Waltz, 1978).

Natural language queries may yield increased productivity from information retrieval (IR) systems. Performance of an IR system is dependent upon the correct use of key words, a function generally performed by an IR specialist rather than the user himself (Shoval, 1981). One can speculate that a natural language system could affect both the indexer and the searcher by providing a standardization of which key words are stored and later generated. The need for a specialist to form queries is removed, and information is made more accessible to the user.

Simply stated, the use of natural language increases efficiency by removing user intimidations, thus encouraging the inexperienced user to take advantage of a machine's power to retrieve data.

1.2. Goals of a Natural Language Processor

It is important to note that developing a grammar that parses any sentence in a natural language would require a machine of infinite resources, while the grammar would possess infinite complexity (Aho & Ullman, 1982). The first goal, then, is to design a grammar that processes an adequate subset of a natural language so that the user is relatively unconfined. Biermann (1982) defines a "fortual language" as:

1. A subset of natural language
2. User learnable from a small amount of instruction

A fortual language is not as powerful as its natural language superset, but it is easier to implement while maintaining a high level of power. In this thesis, when we refer to a system as accepting a user's natural language, we actually refer to a fortual language.

Another goal is that no inaccuracies should proceed beyond the natural language processor into the database system (Harris, 1977a). If an error is passed to the database system, recovery becomes more difficult and often results in the return of inaccurate data. This raises the question of whether an "inaccuracy" should include questions outside of the information base's domain, something often undefined prior to the formal search.

Other inaccuracies should be tolerated. Spelling mistakes should be overlooked or properly corrected where possible. Inferences should be made where ambiguous or vague questions are proposed, and searches should be implemented only after the dialogue with the user has resolved any ambiguities, eliminating any incorrect assumptions. Biermann (1982) believes that narrowing the scope of a database, thereby restricting its domain, will yield less ambiguity and vagueness, while allowing a more refined system to be developed during preprocessing.

The natural language system should provide explicit answers rather than a set of answers. A more developed system would provide natural language responses; a less developed one would provide a table or a list.

Convenience is necessary. The system should be interactive, carrying on a dialogue with the user and the database administrator where necessary. Keeping the system on-line makes the system readily available, and the system should be as quick as possible. Upgrading the system should be made simple by making it easy to extend, both within it's own world and to new domains and databases (Waltz, 1978).

Fundamentally, the natural language processor should hide the underlying database structure. It is not adequate to introduce formal language constructs to the user, nor should it be required that the user have specific knowledge of the information domain. It is most important that all users get appropriate answers to their queries with a minimum degree of instruction.

1.3. Theoretical Requirements of Natural Language Processing

Ballard (1979) lists several requirements necessary to produce a natural language system. These include linguistic knowledge, domain knowledge and programming knowledge.

Linguistics is the study of language which includes grammar, elementary word structures, component structures, semantics, etc. Its role in the design of a natural language processor is to identify the "everyday" limits of the language. If a natural language must be scoped down, to restrict the constructs available, it is important to identify the constructs commonly used by the users for whom the system is being targeted.

Linguistic knowledge directs the designer to an efficient natural language translation of an input query, yielding a highly meaningful internal representation of the user's request. Dictionary entries in a natural language system are extremely important and are influenced by the linguistic knowledge of the designer. A proper perspective in linguistics will generally yield a more useful system.

Domain knowledge is also important. While it has been universally accepted that the more domain specific a system, the less portable it becomes, the domain does influence design strategies of various components of the system (e.g. parser, semantic analyzer). As understanding of natural language processing has increased, so has the ability to produce systems with increased domain content and increased portability. This has been accomplished mostly by taking advantage of the database administrator (DBA) and a system preprocessor (refer to Chapter 3). Nevertheless, in all applications, domain knowledge is a fundamental component of every system.

Programming knowledge is necessary to make control structures available, both to the end user within his natural language, and to a "super-user" within a meta-language. A meta-language is a language that is hidden to an end user, and is at a higher level than the language used for normal operation of the system. It is a language that is used to control the operation of the system and is commonly used to increase the scope of the grammar constructs which define the fortual language being used.

The challenge, then, is to produce a portable, "domain independent" system that can operate over many different domains with a minimum amount of restructuring and no reprogramming.

Of the two most common approaches to building a natural language query system, the natural-language-front-end processor has been more popular than a complete natural language database system (chapter 7), due to its easier implementation. Receiving natural language input that represents a query, a front end processor attempts to translate the input into a formal database query. The resulting formal query is used to access the database, retrieving the information requested by the user (figure 1-1).

The scope of this thesis is to examine the basic elements of a natural language processing system, while describing the implementation of a simple front end processor called NBASE. The major design issues will be addressed by referring in detail to existing alternatives implemented in several current systems (LUNAR, REL, INTELLECT and the LDC-1). Finally, an alternative database approach is presented, describing a database system whose underlying structure and schema are defined by natural language constructs, as well as the access support to the database.

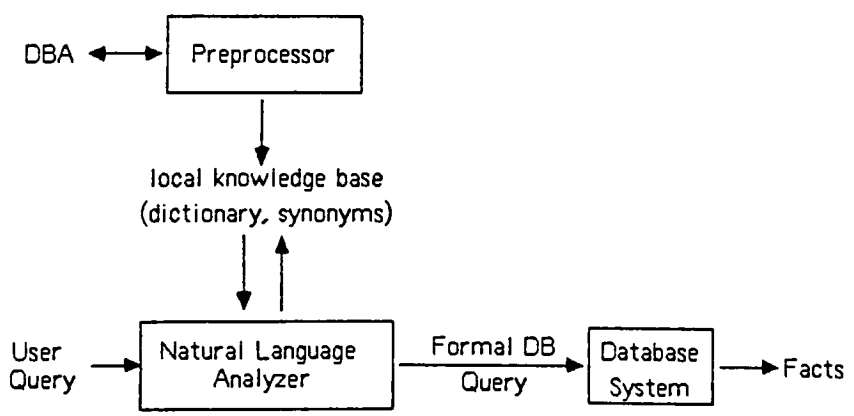


Figure 1-1

The conclusions hopefully will generate interest for future enhancements of the described processor, as well as additional research topics.

2. SYSTEM OVERVIEW

Designing a natural language processor demands considerable forethought and planning. As with any software system, goals must be specified and properties isolated to yield a definition for the processor. Once the scope has been determined, the designer chooses a software architecture, and then a hardware architecture that best supports his software expectations. As we shall see, the properties of the natural language processor require that data structures be developed early. These considerations must be made before a functional "black-box" description can be designed.

2.1. Design Alternatives

Initial design decisions address the issue of goals. What is the application to which the system is to be directed? How portable/flexible should the system be?

Early systems, such as LUNAR (Woods, 1970), were designed according to a restrictive set of specifications. LUNAR's goal was to service the needs of scientists whose information requests were limited to the narrow domain of lunar geological studies. The primary goal was to provide interested lunar researchers with the capability to extract information from an extensive database without requiring training in a formal database query language. While LUNAR was a successful early prototype and pioneered several implementation strategies, its application dependence and non-portability motivated further research toward a more flexible system.

REL[1] (Thompson, 1975) represents an early implementation that addressed portability

[1] Rapidly Extensible Language (REL)

needs. The goals of REL included: 1) The support of many language packages, each with its own syntax, semantics and data structure facilities; 2) factoring the system to separate the driver modules from the application modules; and 3) an extension allowing the user to define new language constructs (via a meta language). REL served as an initial bench-mark for portability. Its designers found they could develop a base system in a given natural language (English, Polish, etc.) and support a variety of user interests by providing an additional "User Language Package." The net result led to a natural language processor with increased flexibility.

Just how portable were these early efforts? Portability is not only a measure of system flexibility, but includes the effort to adapt a system from one domain to another. The REL "User Language Packages" were hardwired packages, requiring recoding for each new application. Gross modifications necessitated the services of a system technician, leading to increased expenditures in time and money.

Harris (1984) believed in the need to strive for a more complete application independence. Asserting that application independence cannot be achieved for the total package, Harris suggested and built a system, called the ROBOT processor, in which all customizations were within the realm of "ordinary" technicians. That is, given a minimal amount of instruction, a company employee could assume the responsibility of modifying the natural language processor to address the domain of any on site application. Harris points out that the ROBOT project was directed at delivering the ultimate in user capability within the context of the existing marketplace and did not necessarily provide the ultimate in AI capabilities. ROBOT has found widespread commercial success and is now marketed under the name INTELLECT. Even the ROBOT system, however, was somewhat limited in its ability to provide full portability because preprocessing that required some intelligent manipulation by the user was needed.

The LDC-1 (Ballard et.al., 1984), though somewhat limited in its scope, represents the state of the art in natural language processing. Like ROBOT, a major goal of LDC-1 was to

eliminate the need for the system technician when a domain increases or an application changes. Ideally, the installation of an LDC-1 system requires no on sight technician with the bulk of the responsibility being placed on the DBA. The LDC-1 takes advantage of the DBA's knowledge of database schema by providing an interactive preprocessor. Prompting the DBA for information as it analyzes the database, the preprocessor isolates relationships, producing a knowledge base that is built around the immediate application. If particular syntactic or semantic information is required during analysis, the local knowledge base serves as the source.

In contrast to most natural language front end processors, the LDC-1 allows for more general retrieval over domains that are not necessarily represented via "restrictive" database structures (see chapter 8). Its capabilities extend to files that are produced by using a file editor, giving more flexibility to the group of people using the natural language system by allowing experienced users to directly modify their stored data. The LDC-1 typifies the research efforts that have been directed at providing flexible and portable natural language systems.

Flexibility and portability represent important research issues in natural language processing. Flexibility is important for each component in the processor to provide transportability and ease of operation, even for the most inexperienced user. Flexibility should be a primary goal of any natural language system to provide robustness for the individual end user and generality for the targeted user group.

Design considerations may include hooks for enhancements or features not originally required in the processor. Ballard and Lusth (1982) suggest several implementation properties that enhance a system's design. The values representing persons (Chapter 4) are difficult to resolve, but by using a series of flags, Ballard suggests alternative strategies to increase the provisions of the processor. Optional properties are most easily defined early in the design phase, removing the necessity for major revisions that occur if these properties are defined during implementation.

An important design feature to establish *prior* to any functional specification is the structure of the operating architecture. This is presented in terms of software, but it could be developed in hardware. The primary consideration is whether or not syntactic analysis and semantic analysis can be separated. Regardless of one's persuasion, it is widely accepted that syntax, semantics and pragmatics of natural language understanding must be well understood before a structural decision can be made[2] (Davis, 1983).

Alternatives in operating structures include a pipelined (or sequential) system and a parallel system. As displayed in figure 2-1, a sequential system emphasizes the separation of syntactic analysis from semantic analysis. The primary advantage of this method is modularization. The syntactic component (or semantic component) can be modified or completely restructured, while being constrained only by the interface between modules. Future revisions of a system may be incorporated by isolating and correcting system components, requiring little or no revision to modular interfaces. While they allow easy revisions, there is a serious question on the degree of efficiency that can be provided by a sequential system. Kaplan

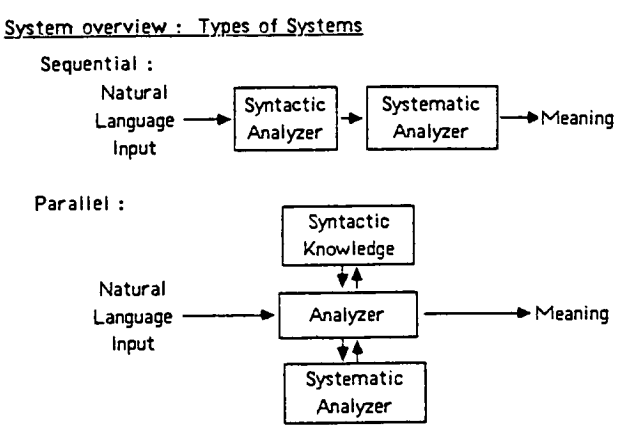


Figure 2-1

[2]The topic of syntax, semantics and pragmatics is very broad, and exhaustive consideration is beyond the scope of this paper. Alternatives are presented throughout the paper addressing particular issues, but it was neither intended nor desired to present a comprehensive survey on this topic. The interested reader could best start with a grammar book on the language of his choice.

confirms that a complete separation of syntactic and semantic analysis may yield inefficient analysis paths. Kaplan believes it is best to incorporate some semantic analysis within the syntax analyzer (Davis, 1983), or parallel to syntax analysis (figure 2-1). Semantic grammars (Wilcox, 1983) also could be defined as a parallel organization.

Arguments against parallel design center around the need to develop a new grammar for each desired domain. The modularity of the system is thereby reduced, supposedly making it less portable. However, if the semantics are resolved in the lexicon (knowledge base) produced during preprocessing, the desired flexibility could be achieved with added efficiency. As flexibility/portability issues are resolved, the emphasis will be on greater operating efficiency, most likely resulting in a more parallel approach.

Mapping a natural language input to a highly related database value makes data structure design the most important aspect of building a natural language system. The data structure not only provides the means for the storage and manipulation of data, but it also defines how the system will work, sets limits on the system's functional capacity, and can be used for code modification to enable the processor to "learn." The strong inter-relatedness between the data structure and the system specification often requires major system revisions if the data structure is modified.

One might simply argue that the data structure is a prime consideration in all aspects of computer science. Thompson (1975) concludes that the short input sentences, the complex interpretive routines and the high relatedness of a database (all characteristic of natural language processing), makes the data structure design central to a natural language processor. When contrasted to the longer input sentences (e.g. entire programs), the short interpretive routines and collections of the otherwise independent data items of formal language processing, Thompson indicates that the natural language processor needs to be considerably more sensitive to data structure complexities and considerably less sensitive to the optimization of working storage.

The research for this thesis confirms Thompson's assertions. Choice of the data structure was central to the design of the entire system. Furthermore, it was found that understanding the process of semantic analysis was the primary influence in data structure design (refer to chapter 6). One cannot overemphasize the relatedness between the data structure and the functional design in natural language processing. Functional design is dependent upon the choice of data representation and the representation is partially defined by the scope of the of the desired system.

2.2. NBASE Functional Structure

Like INTELLECT and the LDC-1, NBASE is designed to provide application independence. The DBA has the primary responsibility for building the knowledge base for each application. A small base knowledge package, providing words such as interrogatives and auxiliary verbs that are common to most natural language requests, is provided for all applications. No other technical personnel are required to ready the system for use by the end user.

The NBASE system provides basic interpretive services for English. There are no additional properties, special flags or representations employed to enhance the system. Although this reduces the system's ability to precisely identify some grammatical errors, it does allow for a more general attempt to resolve any user input. Some general 'hooks' have been provided in the data structure for future additions and hardwired for no particular utility.

The architecture is sequential, allowing for easier implementation and a more elegant approach to revisions. If the data structures are left unchanged, component revisions may be coded readily to provide additional features and/or increased efficiency.

The general data structure is a list which appears to be the most widely accepted data structure for natural language processing (Winston, 1981). The system was built using PROLOG, a logic oriented programming language (chapter 7) that has facilities for convenient list processing. The contents of the list data structure are redefined through each successive stage of the processor, and allow several levels (currently four) of nested structures. The details of the data structure and list processing will be discussed when their respective

modules are discussed.

The dictionary (knowledge base) is the foundation of the system. System versatility is bounded by the level of dictionary completeness, which in turn is established during preprocessing. Dictionary entries include synonyms, word definitions and schema definitions of the targeted database. With the exception of the small base knowledge core, all dictionary entries will vary from application to application giving the system its portability. Learning capabilities are provided to expand the range of the dictionary 'on the fly' (see chapter 4).

The functions of the NBASE system can be separated into three categories (figure 2-2):

1. Preprocessor (chapter 3).
2. English processor (chapters 4-6).
3. Retrieval Module.

The preprocessor, JPREP, obtains the knowledge peculiar to a given domain. The success of a preprocessing session using JPREP is highly dependent upon the experience of the user operating the preprocessor. Generally, the user is the DBA, an individual who is thoroughly familiar with the database schema and the intended relationships among attributes of various relations. JPREP assumes the existence of a relational database, and it yields text files of domain specific 'knowledge.' The text files produced are modifiable, but are generated

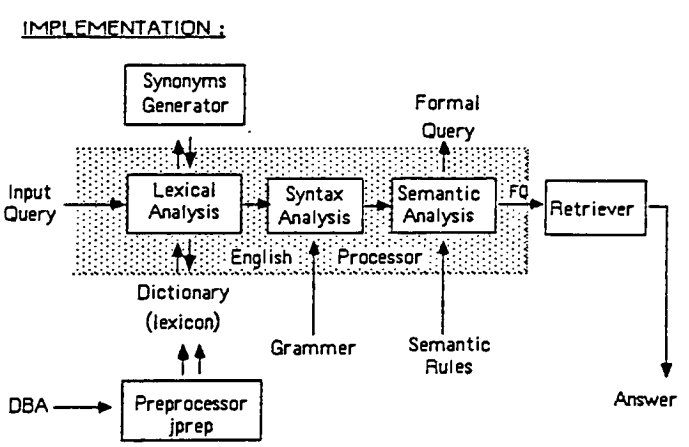


Figure 2-2. System Overview

according to NBASE specifications, and therefore should be protected from all but a system technician.

The function of the English processor is to process the English input provided by the end user, yielding an appropriate formal database query. User input is restricted to keyboard entries of English natural language. The English processor consists of three subprocesses (figure 2-3), the lexical analyzer, the parser, and the semantic translator. As the query is processed, it passes from one module to the next as a series of lists. Each module outputs a derivative of the input list performing translations to ultimately produce a formal query.

The lexical analyzer provides traditional lexical processing (Aho & Ullman, 1978), yielding a list of tokens (words and punctuation) to be processed by the parser. The parser uses an attribute grammar with the addition of global registers (chapter 5) to parse the incoming token list. Some semantic analysis is provided in the parse. This improves the efficiency of the analysis by providing a degree of parallelism as described above (See chapter 5). The goal of the NBASE system is to process as much user input as possible, not to "spit up" errors. With this in mind, the parser is fairly forgiving with respect to grammatical errors. A successful parse results in a parse tree, which is input to the semantic translator. Several

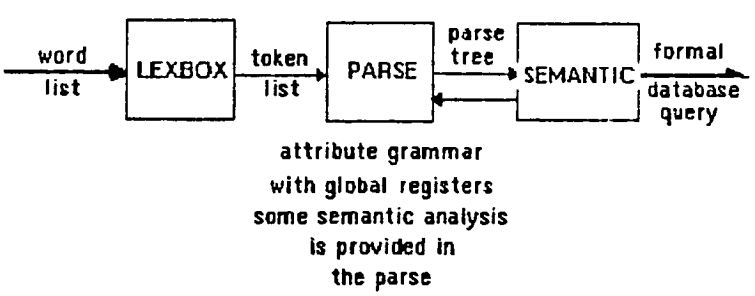


Figure 2-3. The English Processor

heuristics are employed there to attempt to produce enough meaning to develop a formal database command (chapter 6). The formal command generated is database dependent, currently serving MISTRESS (Rhodnius Corp., 1983) on the UNIX operating system (AT&T).

The retriever is a trivial implementation of a PROLOG system call. The call invokes a UNIX shell, which in turn invokes MISTRESS using the MISTRESS user shell interface. No attempt is made to produce a return result in the form of a natural language assertion. The result is supplied according to the specifications of the MISTRESS design. No attempt is made to respond to an invalid query served to the database. It is hoped that most inconsistencies will have been resolved by the English processor before a formal query is issued to MISTRESS, limiting invalid queries. Users should be informed that formal queries requiring the search for specific database values will only succeed if an *exact* match is found, a severe limitation to this system. (See Appendix A).

3. PREPROCESSING AND THE KNOWLEDGE BASE

Until recently, natural language systems that contained enough information to be useful were generally not very portable (Waltz, 1982). Design conflicts often arose, portability opposing system specialization, with portability usually assigned the lower priority.

As natural language packages have become commercially available, however, portability has become the dominant design issue. Natural language processing installations now demand a comprehensive package, allowing for the retrieval of data from numerous domains. Natural language systems rely upon a knowledge base to associate meaning with a query. The ability to restructure that knowledge base to affect portability is the function of the preprocessor.

3.1. The Knowledge Base

Domain Knowledge is the most important component of the entire processing system. Structurally, it delimits the processes of syntactic and semantic analysis by providing word definitions and the underlying database structure to the system. The word definitions of the knowledge base determine what is available and accessible for the current system. Relationships specific to the targeted database are an inherent part of the knowledge base and provide a rich source of semantic information for the front-end processor. Most important, if the semantic relationships can be wholly represented by the knowledge base using a restructurable lexicon, the natural language processor's versatility can be virtually unlimited, and could be transferred to any application (figure 3-1). The chief concern would be encoding the new lexicon.

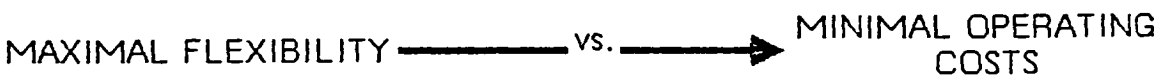


Figure 3-1. Portability Influences

The efficiency of lexicon production is improved by defining an organization over the set of word entries to be encoded in the dictionary. This structure is established by partitioning the set of entries into three subsets, general entries, structural entries and volatile entries (Kaplan, 1984).

General knowledge entries are those entries universal to all domains. They are encoded directly into the system, and form the lexical foundation on which the entire system is built. General knowledge is the domain independent data that can be used with any application. These entries make up the base knowledge package of most systems, including NBASE. Examples of general knowledge entries include conjunctives, interrogatives, meta definitions and auxiliary verbs.

Structural knowledge entries reference the attribute fields peculiar to a database. Should the entries be used to define a system component such as a grammar, then the system would be non-portable. The structural entries are those requiring complete regeneration from application to application and should be created by the DBA. These entries provide the semantics for the system by establishing relationships among classes of objects and defining the level of relationship between objects (close, tangential, etc.). They represent the database schema,

link verbs to nouns and prepositions to nouns, establish synonym lists, isolate attributes, and maintain pathnames, all of which are local to one particular database. Structural entries provide the "possibilities" for query resolution.

Volatile entries are references to specific values in the database. These values are commonly useful only during a particular session and include the special treatment of numbers or numeric data fields, particular character fields, names, locations, etc. They are values that are recognized by some general heuristic and are neither hardwired into the system lexicon nor in violation of database normalization theory. Volatile values are temporarily "learned" knowledge maintained only for the length of the session in which they are defined in order to increase time efficiencies.

Considering the time required to produce a tailored domain that will yield portable formal queries (Leigh, 1983), implementation of a partitioned word set reduces search times costs by concentrating on the generation of structural entries. The design of a preprocessor must balance flexibility and operating complexity/costs to provide optimal lexical generation over any domain/database. Partitioning helps to minimize the cost by minimizing the amount of restructuring required to build a new lexicon.

3.2. Preprocessing

By understanding the structure and generation of the lexicon, we can see that portability and flexibility constraints are set by the preprocessor. Design considerations must focus on both the interpretation of the targeted database and the representation of developed relationships. Interpretation needs to be exhaustive, while representation needs to be well defined. A well defined representation requires that *all* relationships are generated, and that

some ordering over the set of relationships exists that will resolve ambiguous and vague statements should they arise.

The importance of complete representation is illustrated in the following example :

USER: What is the GRADE in table SCORES whose STUNAME is 'Julie Richardson' and COURNUM of ICSP241?

The structure imposed on this input is typical of that required by a formal database query and is not sufficient as a natural language query. It does have a natural language 'flavor' to it, but requiring the user to understand the underlying database structure is a failure to any natural language system. The above query exemplifies the disadvantage of a restricted lexicon and does not promote access for the unexperienced user.

Given a limited amount of semantic information, a less restrictive query would be :

USER: What is the grade for the student whose name is Julie Richardson in the course whose number is ICSP241?

Lexical entries must exist associating the noun 'grade' with the field specifier GRADE; associating the specifier GRADE with the relation 'SCORES', etc. This is an allowable natural language query. However, if the user is *required* to explicately identify structural values (e.g. "... whose name is ..."), the extent of relation representation is not ideal for natural language processing.

The challenge of producing a formal query from a less formal, comparatively vague natural language request, can only be met with the definition of appropriate relationships in the lexicon.

USER: What grade did Julie Richardson get in ICSP241?

This query is typical of a natural language input, requiring no value-structure associations from the user. All associations are resolved using semantic inferences supplied in the lexicon. Assuming a well developed grammar, queries of this sort promote user access by freeing the user from learning the system and understanding the underlying database structure.

The implementation of LUNAR was hardwired, and therefore non-portable and requiring no preprocessing. REL was somewhat portable, its portability defined by a grammar constructing feature, allowing the generation of new domain-specific grammar rules (Thompson, 1975). It required no preprocessing, but its portability was more limited by memory constraints.

INTELLECT, having no preprocessor as such, does employ a level of preprocessing. Each user writes his own lexicon which in turn establishes the boundaries of INTELLECT's domain. Harris claims that writing lexicons is simple enough even for non-linguists (Davis, 1983). However, INTELLECT's ability to resolve database relationships is restricted by the user's concept of the database.

Perhaps the most versatile of preprocessors, is PREP. PREP is the primary learning component of the LDC-1 (Ballard & Lusth, 1982), and builds both the syntactic and semantic relationships for later inferences. PREP isolates the name of each relation in the domain and finds the nature of the relationships among relations. PREP determines the English words to be used (e.g. nouns, verbs, modifiers) in user queries, and attempts to identify the morphological and semantic properties of each word (Ballard, Lusth & Tinkham, 1984).

Specifically, PREP prompts the DBA for associated nouns, adjectives and modifiers for a given entity. These word entries will be used later to help disambiguate meaning structures in addition to representing leaf nodes of the parse tree. Associated with each entity is a synonym list allowing the surface structure to refer to the same entity using several terms. A type is associated, suggesting what the entity might be a reference to, a person, a date, a time, or 'something else.' The DBA can define a unique type via the 'something else' choice by supplying a pattern or context in which the entity might be found (Ballard, 1982).

Verb associations are the most influential of all associations, producing both syntactic and semantic information. Representing a subset of the leaf nodes of the parse tree in syntactic analysis, verb structures are also used to indicate the relationship of a given entity to the domain structure. Much meaning can be extrapolated by cross referencing verb, object and

subject references. The generalities of these relationships are more clearly described in the implementation of JPREP.

3.3. Implementation of a Preprocessor -- JPREP

JPREP is a natural language front-end preprocessor, designed to provide portability for the NBASE system. Code specific details will not be provided here, although a data structure graph is given in Appendix B. Conceptually, JPREP is a large symbol table processor associating word entries with various attributes. It is completely interactive, requiring no understanding of the physical structures it generates.

JPREP's operation requires an interaction with the DBA. The success of JPREP is strongly related to the DBA's understanding of the database and his expertise in defining the relationships between attributes. All relationships among classes of attributes are defined during JPREP's analysis of the database schema and are limited to the internal schema support supplied by the database package.

JPREP offers the capability of reinitializing a complete system or modifying an existing system. All system initializations are local to JPREP execution.

Comparisons can be made relating JPREP features to those found in both CO-OP (Kaplan, 1984) and PREP (Ballard, 1982). Additional features, weakly link JPREP to INTELLECT (Harris, 1978). Three classes of output are produced by JPREP, environmental structures, attribute synonyms, and word definitions. These structures are assumed to be unique to a specific system. Environmental structures are produced through direct interaction with MISTRESS. This is done by executing a UNIX shell script with a MISTRESS command to dump the database structures. After finding the database structure, JPREP uses the resulting information to analyze the relations of the database and prompt the DBA to establish relationships among attributes.

Environmental structures include the location of the targeted database and its related schema structures (tables, attributes, primary key, etc.). The assembled environment data are

used both directly by NBASE (for analysis and retrieval) and to guide JPREP through all database relations.

Attribute synonyms are special synonym structures containing semantic information. The entries associate synonyms to actual database field attributes. Their use reduces the time required to isolate a targeted field reference in a query. The structure provides for both single word and phrase synonyms. Using the relational model in Appendix C, the following example demonstrates the use of an attribute synonym.

```
Synonyms for f# :  family, family number, family #          ==>
        syn_attr([family],f#).
        syn_attr([family,number],f#).
        syn_attr([family,#],f#).
```

Using the generated structures, a later query of the form

USER: What family lives at 24 East Ave in Brockport, NY?

could quickly be analyzed to yield the target field of 'f#', generating the initial component of a formal query,

Select f# from ...

A more general synonym function is supported by the lexical analyzer for user convenience. This will be discussed in chapter 4.

Word definitions are the primary dictionary entries. The entries are generated during a preprocessing session via prompting for words associated with each attribute. The system currently supports four *restructurable* classes of words: nouns, adjectives, verbs and prepositions. Other classes are installed as general knowledge entries at the time of system initialization. The representation of a word is defined to contain four fields of information: the value of the word, the word classification (e.g. noun), a features list and an empty field for late enhancements.

The features list is the semantic information associated with each word, linking a word to the domain structure. Feature values vary from word to word, and the list structure varies from word class to word class. The feature lists of nouns and adjectives are structurally the

same. This list is an encoding of the relation-attribute pair(s) with which the word could be associated. The number of associations has no upper bound, but the lower bound is one, otherwise the word would not be in the dictionary. Noun and adjective entries are built by prompting the DBA for nouns and adjectives associated with the given attribute. As the DBA responds, the words are loaded into a symbol table where relation and attribute identifiers are attached to the word (Appendix B). If a word already exists in the table, it is simply updated to avoid knowledge base duplication.

The structure of verb and preposition features is much more complex, allowing up to three levels of nesting of features. The primary function of each verb or preposition entry is to link each possible subject to each possible object (figure 3-2; the semantic worth of this representation is discussed in chapter 6). This is executed by the preprocessor by first prompting for the verb or preposition and loading the result into the symbol table. The subject values are assumed to be the current relation and attribute identifiers, which are automatically assigned a subject association with the current word. The DBA is then prompted for objects upon which the subject may perform the action indicated by the current verb or preposition[1], linking all objects returned to the subject reference. The number of subject-object references has no upper bound, and also requires at least one per dictionary entry. Duplication of a verb or preposition entry is not allowed.

In contrast to the representation of structural knowledge values, all general knowledge values and the temporary occurrence of volatile knowledge values exist with null feature lists.

Two additional constructs associated with word definitions are the 'who' and 'where' components. Similar to both INTELLECT and the LDC-1 processors, person and place associations are maintained in the system. This is performed by giving the DBA the opportunity to flag a field as representing a person, place or thing. Every person/place field is entered into a who/where list. These lists are used by the semantic analyzer to infer subject-object relationships when a 'who' or 'where' query is processed.

[1]The theory differs for the preposition, but the end result is the same, associating subjects with their related objects. Refer to Appendix D for a sample preprocessor session.

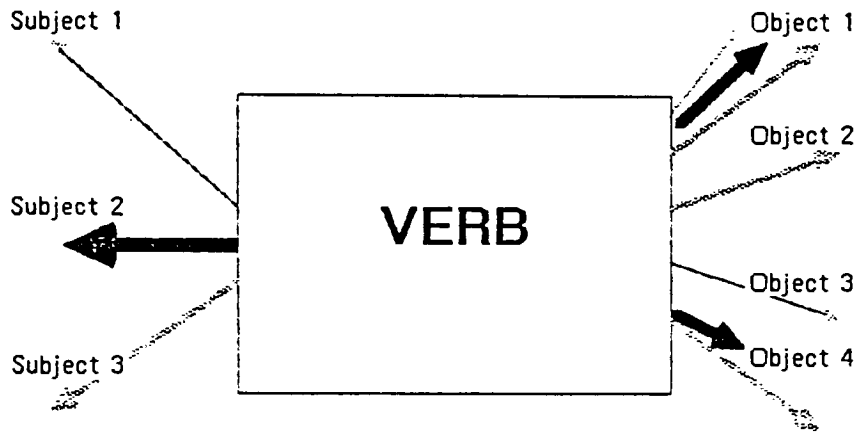


Figure 3-2. Subject to Object Linking

USER: Who lives at 24 East Ave in Brockport, NY?
USER: Where does Julie Richardson live?

To summarize, JPREP is an implementation of a rather simple preprocessor that realizes its full potential for dictionary processing under the current design. Further enhancements may suggest a complete redesign in order to broaden the scope of JPREP. Testing of the NBASE system has shown JPREP to be an adequate preprocessor for achieving application independence that realizes its primary goal which is to provide portability.

4. LEXBOX

Lexical analysis is the first stage of compilation, representing the interface between the source program and the compiler. In natural language processing the source program is the natural language query. The role of a lexical analyzer seems to be much less complex in a natural language implementation than in its formal language counterpart.

The basic function of a lexical analyzer is to scan an input string, breaking the string up into logical entities called tokens. A token has the general structure of a 'type'-'value' pair, where type identifies the lexical class of the input (e.g constant, identifier, key word) and value represents the actual input received. For example, given the following Pascal statement:

HORSE := 5;

a possible token output of the form (type, value) might be

(identifier, HORSE)

(operator, ':=')

(constant, '5')

Token organization differs among implementations with most organizations being more complex than the above example.

Depending on the lexical-syntactic analysis interface, tokens are either output one at a time, or organized into a list structure and output as a token list. Assuming the latter, the lexical analyzer's general function is to convert an input string of some source language into a token list (figure 4-1). The lexical output then serves as input for syntactic analysis.

Lexical analysis is important to compilation for several reasons (Aho & Ullman, 1972). The most significant is the resulting processing convenience produced when replacing

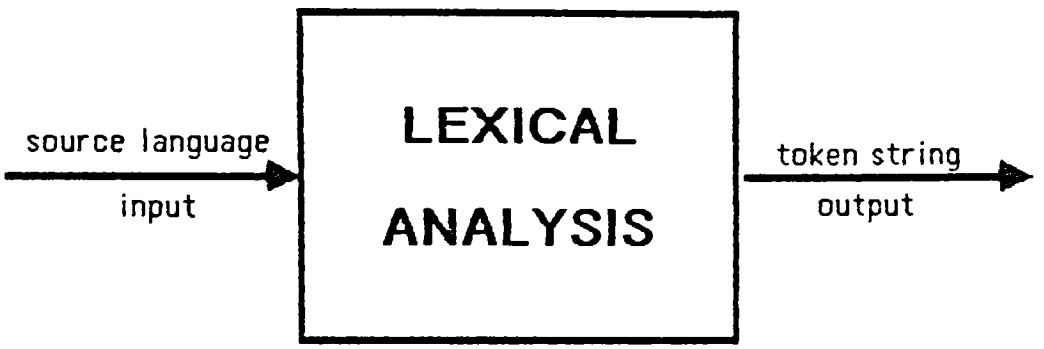


Figure 4-1

individual source program atoms with general, logical tokens. Lexical analysis reduces the logical length of a source program (or string) by removing extraneous data such as blanks and comments. By dividing the overall analysis into two stages, lexical and syntactic, the overall design is simplified, providing a degree of modularity to the system.

The LEXBOX of the NBASE system does the typical lexical analysis described above. The source program is a natural language input, and the lexical function is to convert the input into a token list. LEXBOX indirectly provides other features somewhat unique to natural language, as described below. Note that symbol table searching, a common function associated with lexical analysis, is performed by searching the associated dictionary.

Initial processing begins in LEXBOX at the input procedure. The primary function of the input component is to capture user input and convert it to a list of atoms called words. A word is any non-null substring of characters, delimited by blanks or the following set of punctuation characters {comma, period, question mark, exclamation point, semicolon, colon}. With the exception of the comma and semicolon, these characters signal the end of a sentence and terminate the lexical process.

All substrings beginning with a capital letter invoke a special search over the dictionary, attempting to match the incoming word with an upper case or lower case entry in the dictionary. If the corresponding entry is not found in the dictionary, the substring is assumed to be a volatile knowledge value. Volatile knowledge values often can be identified by proper nouns. Other volatile entries are caught through interaction with the user if an incoming word cannot be located in the dictionary. As such, it is temporarily asserted into the knowledge base as a noun, and given a special volatile feature list. Regardless of the success of the dictionary search, the isolated atom is inserted into an input list.

The primary lexical analysis stage receives the resulting list of words as input. The first processing of the list is to identify the logical entity to which each word is associated. There are three token types generated by LEXBOX, punctuation, integer and word. If a substring is identified as an integer or punctuation, the corresponding token type is generated, and the substring value is linked to the token identification. As tokens are constructed they are appended to the token list representing the current user input. The token list is the data structure to be output from lexical analysis (Appendix E).

Identification of token type 'word' is a more complex and time consuming process. A word token association is made only if punctuation and integer analyses are not successful. Word analysis requires a dictionary search for each word. If the word is upper case, both upper and lower case searches are attempted. If the search is successful, a word token is constructed, and several values are associated with the token. A design choice was made to associate information to be used in later syntactic and semantic analyses with the word token, to reduce the amount of processing time required for additional dictionary searches. The tradeoff is higher memory requirements, a resource considered to be far less expensive than response time throughout NBASE design. The values associated with a word token are its value, its class (noun, verb, etc.) and the associated features list.

An unsuccessful dictionary search is indicative of two possible failures. Either the user has misspelled the word or the word does not exist in the dictionary. A feature not provided,

but left for a future enhancement, is a spelling checker. It is at this point in the analysis process, that a spelling checker would be invoked and an attempt made to resolve the word reference. It would be important to design a spelling checker that would allow the user to interactively approve all processing assumptions.

If a word is still not resolved, a learning component has been provided to interactively work with the user to produce a dictionary entry for the new word. The user is first asked if the word under analysis is a database value. If so, a dictionary entry is built and the substring is temporarily asserted into the dictionary as a volatile value. If, however, the substring is not a database value, the user is prompted for a list of synonyms related to the substring. The synonym list must be a non-empty response.

Receiving a list of synonyms as input, the synonym generator searches the dictionary for all occurrences given in the list. If no match occurs, the user is prompted again for a list of new synonyms, this cycle continuing until at least one match is found. When a match is found, the characteristics of the matched dictionary value are attached to the new word. A dictionary entry is created and permanently loaded into the knowledge base, creating an illusion of learning. The dictionary update extends over all future processing sessions, not just the related active session.

When either a spelling checker or the synonym generator is successful, the final process generates a word token with the connected properties defined above. The token is appended to the token list for later analysis (figure 4-2).

The implementation of LEXBOX is recursive, and can be invoked if a declaration is used in response to a prompt from LEXBOX. LEXBOX will prompt the user if it cannot resolve a reference and is not sure what to do with it. Allowing natural language input, the response also must be analyzed, requiring LEXBOX to be recursive in order to properly interpret the response. An example is the failure of LEXBOX to identify an input word. The word may represent a volatile knowledge value, but is not a proper noun, and, therefore, not capitalized. LEXBOX will then prompt the user for either a synonym list or a determination

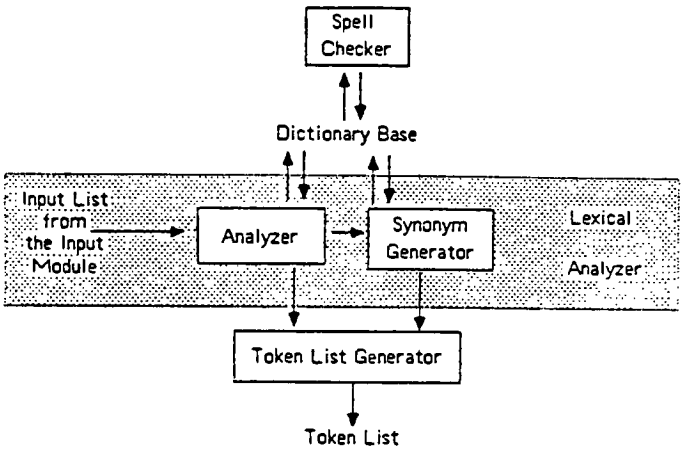


Figure 4-2

of the word's database value. LEXBOX must be able to analyze any natural language input that requires recursively calling itself. (Note: The hooks for this feature are provided. Currently, LEXBOX prompts the user for a response within the format expected by LEX-BOX. The recursion is in place, however, it is the grammar that is not complete.) Currently, the nesting limit is not software defined. A valuable enhancement would supply an escape if some meaningful threshold were identified, indicating over-commitment to the recursive process.

Of the systems examined in the literature, only LDC-1 displayed a separate lexical component, although the lexical function was integrated into various modules in all other systems. The use of a separate lexical tool may not be considered a necessary component in natural language processing, its operation being somewhat less complex than that of formal language analysis. NBASE research, however, proved the lexical analysis phase to be worth implementing as an individual component, providing a theoretical base on which to build the subcomponents featured by it. It also delivered a meaningful approach to theoretically describe the system, and it made the overall system implementation simpler.

5. SYNTAX ANALYSIS

Syntax analysis is the process that defines how language expressions can be generated. It is the method by which valid sentences are recognized/built. Accepting a list of tokens generated during lexical analysis, the parser examines the token types to determine whether or not the string satisfies the structural rules explicit in the syntactic definition of the language (Aho & Ullman, 1972). As the token string is processed, the parser imposes a tree structure (not always explicitly) on the tokens, which is used in succeeding stages of compilation.

Using a Pascal substring as an example

A + * B

lexical analysis may generate a token string to make the Pascal input appear as

identifier + * identifier.

The parser should detect the error of two adjacent binary operators, an illegal string according to the syntactic definition of the language.

The second phase of syntax analysis imposes some hierarchical structure on the processed token stream, grouping related tokens together. The generation of such a structure (the parse tree) often leads to an ambiguous production of multiple structures, requiring the language specification to disambiguate the result. For example, the expression

A * B / C

has two possible interpretations.

A. Multiply A and B, then divide by C.
B. Divide B by C, then multiply by A.

The two interpretations are represented by two different parse trees (figure 5-1). The challenge of developing an unambiguous language specification is a primary concern in the construction of any parsing system[1].

Parsing natural language is even more challenging. The ability of a computer to accept a sentence in a natural language, such as English, and recognize its grammatical form has been tested for nearly three decades with varying degrees of success. Several solutions have been proposed, some directed at efficient performance, others presented as answers to linguistic and psychological questions (Wilcox, 1983).

Before examining some of the parsing strategies developed particularly for natural language analysis, we must examine one of the most influential concerns, the problem of ambiguity in natural language.

5.1. Ambiguity

Ambiguity is more dominant in natural language than in a formal language. As noted earlier, ambiguous representations in a formal language can be controlled by restricting the

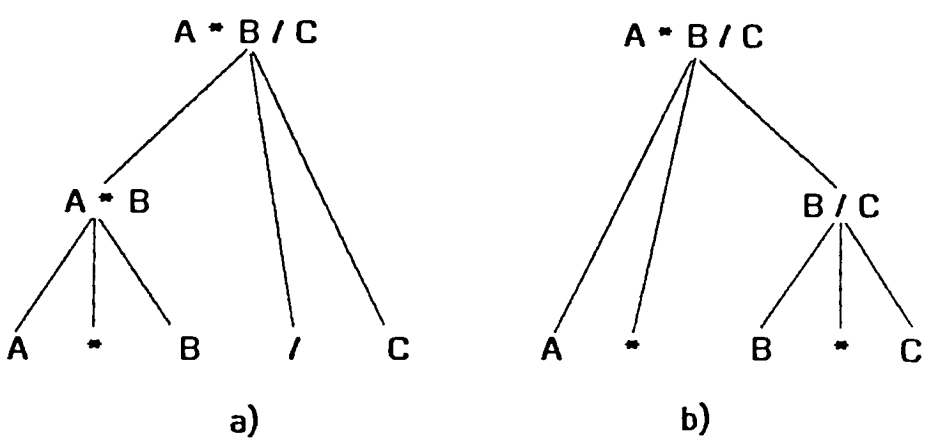


Figure 5-1

[1]For a more detailed study on the formalities of parsing, its strategies, techniques, etc., refer to Aho & Ullman, 1978.

language specification. This "convenience" is not usually available to the designer of a natural language parser. Recalling the goal of a natural language processor, to exploit the user's knowledge of his natural language, restricting user input to avoid the ambiguities of the natural language is not acceptable. Some English ambiguities can be disambiguated by considering the context within which the related input is found. Often, the selection of an interpretation is made by the user, choosing from among several alternatives given by the system.

Syntactic ambiguity in natural language parsing emerges from the fact that "modifying clauses" are often separated from the structures they modify (Kaplan, 1984).

The man drove down the street in a car.

Is the street in the car (a valid parse) or is the car on the street? Kaplan has suggested the following heuristics for resolving ambiguities, but stresses that syntactic information, itself, is generally not enough.

Kaplan's first suggestion is to define a measure of distance over the components of the surface string. Distance is a physical calculation of the position of all potential noun phrases in the sentence. Potential noun phrases are ranked according to their order in the input, the first position receiving the higher rating. Using the previous example, "the man" would be the leading noun phrase in the sentence because it is the first noun phrase encountered and until otherwise determined is assumed the subject of the sentence. Now we can generate the correct way to produce this parse. It is

The man who drove down the street - and
The man who drove in a car.

While the decisions affecting the proper parse for the above example required no semantic information, we could modify the surface structure of the string to produce a construct that would yield unacceptable results when applying Kaplan's heuristic. A supplementary proposal is to add an additional feature to the lexical entries of all verbs and prepositions indicating the proper head association when more than one alternative exists. A head

association links the verb to its subject-object reference(s). When an ambiguous parse is detected, a combination of distance measure and a comparison against the features list would help to isolate the correct structure. Again using the above example, the verb "drive" would have a subject association relating it to the man, with further listings identifying both "street" and "car" as potential objects. If no association is built between "street" and "car", the proper perspective is obtained (this is not always a possibility).

If necessary, relatedness can be further measured by examining the semantic closeness of more than just the surface string. The set of final parse structures can be reduced by measuring the distance between the database schema structures targeted for inquiry, provided that such information is available. This measurement is based on a ranking of schema entities, assigned according to their relatedness or closeness in the structure of the database. Parse structures are then accepted using the highest rank in the deeper structure which is similar to the methods employed in handling the surface structures. This alternative is commonly used with the previous two as a final attempt to isolate one parse structure.

Harris (1978), demonstrated further ambiguous peculiarities using INTELLECT (ROBOT). Questions requiring the combination of syntactic information with domain specific information are perfectly valid, yet difficult to build a parsing structure for. As an example, consider the relationship of time referrants to periodic database fields.

"In 1972, what were the profits and total sales?"

The problem is grouping 1972 with the proper modified construct. Should it be associated with "profits", "total sales" or both. Harris seems to agree with Kaplan, suggesting the answer should be resolved by using additional dictionary specifications to decipher the ambiguity.

Furthermore, Harris raises questions directed at systems able to recognize certain abbreviations, or requests for data that are represented in a coded form. For example, the ME or IN problem:

Indiana vs. IN

Maine vs. ME

How should the analyzer interpret when a string using these forms is scanned? Does IN represent a preposition?; ME a pronoun, or are they coded data associated with a particular database field (like states)? The designer of such a system is challenged twice: first, to realize that asking the user for more information may display a degree of triviality in the system not appreciated by its users, and second, the need to maintain domain independence. Too many limitations are built into a system that handles this particularity, at the expense of portability.

Some heuristics have been suggested, such as those discussed above, to answer several problems associated with ambiguity. There seems to be no one algorithm yet accepted by the natural language research community, but one central thought does emerge. The heuristic used must support portability, causing most of the research to be centered on modifying the lexicon. The accepted approach to resolving ambiguity called the generate and test approach is to (Winston, 1984):

1. Generate all possible parses.
2. Narrow the alternatives to one, using a portable heuristic.
3. If all heuristics fail, let the user isolate the structure.

Resolving syntactic ambiguity only solves part of the problem. Semantic ambiguities that further complicate the process of natural language analysis are discussed in chapter 6.

5.2. Parsing Strategies

The strategy employed in parsing a natural language is more complex than that of a formal language, requiring a more powerful grammar or recognizer. Several tools have been proposed during the last two decades, some of which have been proven to be lacking in power.

A system developed by Kuno (1962), the multiple path syntactic analyzer, used a context free grammar to attempt to generate all possible parses of a natural language input. Context free grammars are those most commonly used by today's formal language processors. Kuno's system helped to demonstrate the shortcomings of using a context free grammar for natural language analysis. Besides an unwieldy grammar of 3400 productions, the parsing

algorithm could become very time consuming, particularly when analyzing ambiguous sentences. Several parses could be generated that made no sense at all because of a lack of context sensitivity unavailable in context free grammars. It also can be shown that not all English sentences can be constructed using context free grammars (Wilcox ,1983), further limiting the scope of this processor.

The most widely accepted strategy for natural language parsing is the augmented transition network (ATN). An implementation of a transformational grammar (Chomsky, 1963), the ATN is a combination of a finite state network, recursion, and a set of registers used to save context sensitive information. Developed by Thorne (1968), it was Wood's (1970) implementation for the LUNAR project that represents the benchmark for ATN processors. With the power of a Turing machine, the ATN has the ability to generate any English sentence.

Its basic operation is that of the recursive transition network (RTN). The RTN attempts to generate a parse structure by processing constituent structures of a sentence. If one particular construct appears nested within another (e.g. a noun phrase within a noun phrase), recursion is used to resolve the innermost construct. When the innermost construct is resolved, its resolved structure is "popped" back to the higher level process, where processing is resumed (see Wilcox, 1983 for an example).

The addition of registers to the RTN allows for semantic information to be collected during the processing. The data collected in the registers may be used during later analysis to organize the resultant parse structure and check associations such as plurality constraints, tense, etc.

Wood's implementation included several enhancements over the original work done by Thorne. Included was the ability to perform both breadth first and depth first searches (an interesting feature, commented on later in this chapter) and several optimizations, including a well formed substring table (WFST). The parsing method was most closely related to a recursive descent parser, with the ability to maintain a list of alternatives if more than one path was traversable in the network.

It is interesting to note that REL, INTELLECT and the LDC-1 all use the ATN in their parsing implementation. Though other methods were experimented with, the ATN was long accepted as the most powerful tool available for natural language parsing.

Recent research, however, has produced another parsing implementation that is gaining respect in natural language research. The "Wait And See Parser (WASP)" is similar to an ATN in that they both recurse down to sublevels for sentence component recognition, but they differ in two major respects. The first is that a WASP employs a look-ahead strategy as opposed to the backtracking strategy of an ATN. Using a three level buffer the WASP is able to predict the sequence of actions to be taken when a given structure is scanned. This eliminates the necessity for retranslation that occurs in a backtracking method when failure occurs. Secondly, the WASP employs a noun phrase preprocessor which identifies and groups all noun phrases together before the main parsing algorithm is called. The main parsing algorithm is built to look for "prepackaged" noun phrases, rather than to identify the subcomponents of the noun phrase. An interesting feature of the noun phrase preprocessor is its ability to correctly identify and group nested noun phrase constructs, which requires a degree of semantic processing. (Refer to Winston, 1984). It is also interesting to note that the noun phrase preprocessor is usually implemented as an ATN.

Winograd's SHRDLU program (1972) was an implementation of a systemic grammar and was instrumental in integrating syntax and semantic analysis. The systemic grammar is as powerful as an ATN but emphasizes different aspects of a natural language. The systemic grammar was based on the belief that sentence organization could be determined by the function of the sentence and the information it was intended to convey.

Several other approaches have been used, attribute grammars, case grammars, slot grammars, employing many strategies, top-down, bottom-up, deterministic (WASP, PARSIFAL), non-deterministic, etc. Whatever the method, the parser is a natural language component that deserves much thought and consideration. A limit on the power of the parser is regarded as a key constraint which limits the entire natural language processor.

5.3. The NBASE Parser

The NBASE parser was designed as an experiment, rather than an attempt to satisfy the demands of the commercial market. The generator used is an attribute grammar which is a context free grammar that passes back a limited amount of information as the parse progresses (Clocksin, 1981). Conceptually, the grammar passes back the necessary information associated with each token to aid in the semantic analysis. Examples are database values associated with a qualifier field, feature lists of the object, subject, verb and preposition, and sometimes the word value itself. Occasionally, an error message is passed back up through the parameter list. The error is loaded into a register to be later transmitted to the user if all parsing paths fail. In addition to parameters passed up, a set of registers is maintained to isolate the subject and the object, as a more global reference. These registers parallel those used for an ATN, and are easily set at a high level of analysis to reflect the deeper structure of the surface string.

The attribute grammar was chosen for two reasons. The first reason was to compare its efficiency with that of an ATN. Testing was not provided to determine the amount of cycles or cpu time required by the implemented grammar and an equivalent ATN, primarily because of the lack of a comparable ATN. Response time on a reasonably unloaded system, however, was adequate. Like Kuno's system (1962), the grammar did tend to grow rather quickly, requiring an additional high level production for each minor deviation of an input structure. As an example, the two sentences

1. Where does Julie Richardson live?
2. Where does Julie Richardson live during the summer season?

required two high level grammar rules to parse correctly, even though sentence #2 contains only the added preposition. Although the existing grammar could have been developed more efficiently (see end of chapter), the ATN, which better handles the processing of nested structures, would have yielded a more efficient implementation of this type of sentence organization. This proved to be a predominant characteristic throughout the grammar.

The second reason the attribute grammar was chosen was because of the apparent implementation convenience built into PROLOG. PROLOG supports a grammar notation (Clocksin, 1981) which is ideal for developing an attribute grammar. PROLOG also provides a built-in backtracking capability (which can be shut down) and argument passing which, together with the grammar notation, provide a powerful and easily understood parsing implementation. However, an ATN could be easily constructed using the same features of PROLOG. The backtracking facilities of PROLOG make the maintenance of global ATN-like registers tricky, requiring registers to be implemented as parameters, or requiring special register manipulation routines to eliminate multiple representations of the same register. Global representations are not erased in PROLOG backtracking, but parameters are. However, efficient design of a register maintenance function makes register processing of ATN information trivial. An early revision of the NBASE system would replace the attribute grammar with an ATN, supplying more power to the system, and producing a less cumbersome parser.

NBASE research revealed that successfully partitioning the operation of syntax analysis and semantic analysis into two distinct components is a non-trivial task, if it is possible at all. This finding supports the assertion made by Kaplan, which questions the effectiveness of a natural language system that completely separates syntax and semantic analysis. Initial attempts with the NBASE parser to provide strict modularization of the two processes failed to produce the returns expected of the system. Therefore, two modifications were made to the original "syntax only" design of the parser to increase efficiency.

The first modification was designed to aid in later semantic analysis, by adding a degree of context sensitivity without directly adding semantic checks to the parser. This was achieved by adding the ATN-like registers mentioned earlier. The only two registers currently maintained (not including a non-standard error register), monitor the deeper structure of the parsed string via correct assignment of the object and subject of the sentence. Particular key words (often called agents) have the property of reversing the position of the subject and the object in a sentence. Since it becomes very important during semantic processing to isolate the proper subject and object references, an accurate parse is imperative.

An example from Wilcox (1983) illustrates :

"The river was crossed by the troops."

In most parsers, including the LUNAR ATN and the NBASE parser, the first isolated noun phrase in the input is assigned to the subject "register." In this case, river is initially identified as the subject. The heuristic applied to an input in which a component is found to be out of sequence, as when the object precedes the subject, varies from implementation to implementation. Using the grammar designed for NBASE, it was trivial to identify a grammar rule containing such key words as 'was' and 'by', and simply reverse the identified subject and object references. Reversal is executed only when the rules prove successful. Unfortunately, the ability to identify such a reversed sentence structure was made at the expense of enlarging the high level implementation of the grammar.

The second modification to the "syntax only" design used semantic information more directly. It was found that simply parsing the input into its component structural parts required duplicate processing in some phases of semantic analysis. Using, an idea presented by Ju (1984), the system was changed to enable the identification of the database target attribute during parsing. The target attribute is the field in the database that contains the information being requested by the user. Using the synonym-attribute list provided by the DBA during preprocessing, the grammar would seek to isolate field and table references for inquiry. Having isolated the target reference, all that is required is to identify whether the target is the object or the subject, which is done using the previously discuused modification. By identifying targets early in the analysis, dictionary searches were reduced, increasing the time efficiency of the system, not to mention the decrease in the amount of code. There are some cases in which the target attribute is not identifiable during syntax analysis (see below). In such situations multiple target values can be identified using recursion. The NBASE parser does, therefore, handle a query such as,

"What street, city and state does Julie Richardson live in?"

isolating as target attributes the fields associated with street, city and state.

Another heuristic employed identifies the word entities to be used as qualifier values during database search. Qualifier values, are those word entities used in the formal database query against which a match is being attempted (chapter 6). Using the volatile flags, the parser is able to identify the *qualifier value* and pass that information on to the semantic analyzer. The *qualifier attribute* is not identified during syntactic analysis! It is the product of extensive semantic analysis.

Four major components are sought during the parsing process. Already discussed were the subject and object, which technically, represent the target attribute and the qualifier value. There are some sentence types accepted, however, that cannot isolate a target value. Who and where questions are the leading examples. Using the previous example, a subject reference can be assigned to "what", but the target attribute cannot be isolated. The process of isolating a target value, in these cases, is semantic oriented, and processing is delayed until the semantic analyzer is called.

The remaining components are the verb and the preposition references. The verb is the primary reference linking the subject to the object, and is the component through which most meaning is derived. Its syntactic recognition is done at a high level, and all semantic content is resolved in the semantic processor.

The preposition, generally represents further refinement of the search pattern on the database. Additional modifiers in the form of qualifier values, are produced to eliminate the amount of information passed back to the user from a formal database query. All secondary qualifiers are handled the same way as the initial qualifier, returning the string value, but not isolating the qualifier field in the parser. The significant difference is that nested qualifiers are resolved in the preposition, allowing for unlimited refinement of a query.

Occasionally, the initial qualifier is buried in a preposition as in:

"Who lives at 365 North Main St.?"

When this occurs, the initial qualifier is set to null and indirectly sent through the preposition

structure. Later semantic analysis uncovers the detail and processes accordingly.

The parser's ability to handle ambiguity is somewhat limited, since no heuristic has been implemented in the current parser. This represents a linguistic limitation, but few restrictions from the view of implementation. The power of the parser to resolve target attributes and qualifier values enables generation of correct meaning to provide the best formal query in most cases. If ambiguity does present a unresolvable problem, the user is presented the alternatives from which to choose.

There is one major deficiency in the current grammar design. Using a search tree as a method of displaying the design strategy of the grammar (figure 5-2a), the corresponding tree is too broad. That is, instead of delaying recognition decisions to later depths (figure 5-2b), the parser attempts to isolate a branch earlier in the tree. This reduces efficiency by repeating previous successes that could have been grouped together in a higher level 'ancestor' node in the tree. Besides speed efficiency, the user is most directly affected by the grammar's inability to flag the *particular* error that caused an unsuccessful parse. The flagging of an error in a broad decision tree must be delayed until no match can be found. At that point isolating the error that caused the closest match to fail is difficult. General error messages have been provided, giving somewhat less than user friendly service.

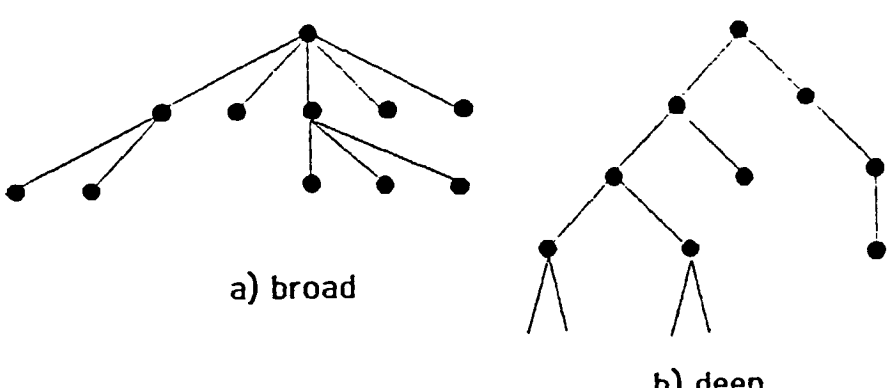


Figure 5-2

6. SEMANTICS/FORMAL QUERY GENERATION

For decades scientists have sought to develop an electronic system that could emulate the thinking process of man. One concentrated effort, has been in the field of natural language understanding. Deriving meaning from a sentence or a list of words is a complex procedure, not yet fully understood. Persisting through many setbacks, research into natural language understanding continues with limited success, still providing an ample frontier for further study.

The formal language process of meaning derivation is exemplified in code generation. As a syntax analyzer is able to provide a successful, disambiguated parse, the resulting parse tree is analyzed to generate code understandable to the machine. Relative to natural language understanding, a formal language provides an ideal representation for the automatic generation of meaning. We can identify several properties of formal language semantic analysis that are not necessarily present in natural language semantics. The elementary formal language constructs are associated with a limited set of meaning, and though they may be combined to generate an infinitely countable set of meaning constructs, such meaning is generally well defined or not a meaning construct at all. The best designed formal language is one with a well defined semantic description (Horowitz, 1984). Generally, a one to one function exists, mapping a fundamental language construct to its corresponding meaning. This is not true in natural language, indeed the ambiguity in natural language does not allow a function to be declared mapping one natural construct to *one* meaning component. One word may have several meanings, and the complexity of meaning generation for a list of words is even more profound.

6.1. Vagueness and Semantic Ambiguity

Semantic ambiguity and vagueness are two of the leading research topics in natural language analysis (Kaplan, 1984). Semantic ambiguities are the most difficult to resolve, usually requiring substantial interaction with the user. As mentioned previously, multiple meanings associated with a word allows one parse structure to be interpreted in more than one way. For example,

USER: Where is New York?

generates one parse structure, but immediately we ascribe two definitions to it; New York as a city and New York as a state.

Some of the same heuristics applied to resolving syntactic ambiguities may produce successful results if applied to the semantic uncertainty. Subject-object predictions from the features list may suggest how an ambiguous term is to be interpreted. Examination of adjacent word definitions in the parse tree, may direct the semantic process to favor one representation over another. However, it is unlikely that either of these two heuristics would resolve the conflict in the 'New York' example above.

Searching a history list (see "vagueness" below), may contextually indicate how 'New York' has been used, from which a reasonable guess can be made. This will require user confirmation, but unless the system is made aware of the conflict, the best solution is to ask the user how he intended to use the questionable term by presenting the available alternatives.

Generally, unless it is strictly interactive, the best automatic resolution, of any ambiguity, is based on strong representation of semantic relationships (Rhamstorf, 1979). Measuring semantic relatedness assumes a high degree of semantic content, a demand that may restrict the portability of some systems.

Some ambiguities may develop as a result of particular system implementations, as in the ROBOT processor. The ROBOT knowledge base includes representation of database target information, organized into subsets of various completeness. ROBOT manipulates the under-

lying database by structuring the relationships of the database schema into several files. As a query is produced, ROBOT attempts to resolve it by using a file of reduced size but with less detailed information. This helps reduce system overload, markedly reducing search times. However, some queries may require more detailed knowledge than is provided by the current file. If necessary, ROBOT generates a summary, which adds more specific data to the knowledge base, using a larger file. Ambiguity is introduced following a summary when a query is given that could be answered using the existing summary, or a configuration that has since been updated with additional information. Query resolution can be accomplished either by searching the existing summary (which is the most expedient, not necessarily the most correct) or by re-summarizing which is very expensive. ROBOT's solution is to provide additional semantic descriptions within the dictionary of summarized information The added descriptions help to isolate the proper algorithm, reducing error and resource overhead (Harris, 1978).

Semantic vagueness results when the topic of conversation is assumed by indirect reference from past dialogue. Concepts may be left out of a given sentence, requiring inferences be made to supply missing information. This is illustrated in the following example:

USER: How many students received a score of 80 or better
on the ICSS201 final exam?

Following an appropriate response from the targeted source of information, the user may then ask:

USER: What are their names?

Most users, would typically associate the word 'their' in the second query, with the 'students' of the first query. Such associations are not done trivially by an electronic language interpreter, if they can be done at all. The derivation of any meaning from the second query requires a facility providing semantic context sensitivity.

The application of context sensitivity does not represent a turn-key resolution of the problem, either. How far back into the context should analysis proceed before an inference

is made? There are several "distances" associated with the assumptions of communication; inferences can be made from the current sentence, the previous sentence, the current paragraph, sometimes from the entire text. Guaranteeing the proper inference can be done only by user confirmation.

The LDC-1 attempted to resolve the vagueness associated with natural language queries by maintaining a history of constituent sentence structures. Evaluation of a vaguely expressed query proceeded by applying various inference techniques on the history table (Biermann, 1982). The history represents the context of the previous dialogue, from which missing information can be acquired.

The advantage of a strong relatedness between semantic and syntactic analysis in the LDC-1, is displayed in the process of resolving a vague query. It is the parser that isolates the existing constituents of the current query. If a vague semantic gap is detected, the parser is called in an attempt to fill in the gap using the inference techniques cited above. If a result is produced, the user is asked for confirmation. If the proper inference was not made, the procedure continues with the next closest reference. If a result cannot be obtained, the user is asked to expand the query, or it is thrown out.

Resolving ambiguity, and vagueness constraints are issues that need to be addressed in any natural language system because of their widespread occurrence in natural language. What makes a natural language front end processor difficult to implement, however, is the feature of portability. Representing all relationships over all possible domains, encompassing many different meanings, is an effectively infinite task. The generation of just those relationships required to manipulate a particular domain is the function of the preprocessor discussed in chapter 3. The next challenge is to implement a general semantic analyzer that will manipulate any knowledge base, and is able to make inferences from the details of the lexicon provided.

6.2. Implementation of a Semantic Analyzer

The degree of difficulty in semantic implementation increases with the amount of portability designed into the system. In a hardwired system like LUNAR, a semantic table may be constructed which is a table associating particular processing alternatives with key words matched in the query. Such a hardwired system should contain a well defined semantic description, one in which inferences can be directly made from the constituent structures of the incoming query. Semantic ambiguity is reduced and ambiguous references are resolved by limiting the meaning of a term or phrase to that of the targeted application.

Portability, however, requires domain independence. Tables associating key words with processing alternatives are inappropriate, except for the most general terms. A general term is defined as one common to any query regardless of the domain, such as, who, what, where, when, etc. The function related to such a key word is an upper level function, one in which domain dependent processing is buried several levels down, yet one that is required to direct the lower level processing along a particular path for complete resolution.

One of the most difficult jobs during the design phase of the NBASE semantic analyzer, was choosing a data structure best suited to domain independence. A versatile data structure was required, one that could be manipulated like a table, yet easily constructed at preprocessing time. As with most natural language processors, the data structure is a fundamental construct, particularly during semantic analysis. NBASE research proved the semantic processor to be the most influential determiner of the choice of a data structure. The wise implementor would best design the data structure around the requirements demanded by semantic analysis before any other facility is examined. A data structure that is not properly designed usually requires considerable changes to all higher level functions and structures. Semantic analysis is considered to be a low level operation (in NBASE, the lowest).

Using the list structure provided by PROLOG, the logical data structure links objects and subjects together, using the verb and preposition values found in the query. In most cases a target field can be isolated during the parsing phase of the process. However, finding

a qualifier attribute, which is a field used to narrow down the scope of a database search, is not directly derivable.

With each associated noun (adjectives work the same way) in the dictionary, is a list of associated ordered pairs:

$$\{(R_a,A_b),...,(R_x,R_y)\},$$

where R = a database relation

and A = a database attribute within R

Each ordered pair identifies a particular field in the database that the related noun could reference. Notice that there is no limit to the number of fields to which a noun could be related. If a field were found to exist in more that one relation of the database schema, it is assumed the noun referenced *all* occurrences, and an association is built automatically, rather than asking the DBA. This assumption is valid because of the constraints placed on a relational database (Date, 1983). It should be understood that, the associated fields are references to nouns rather than subjects or objects. This allows any noun to be in any position, subject or object.

The verb representation is more complex. Attached to each verb is a list of lists:

$$\{ \text{S-OL}_1, \text{S-OL}_2,...,\text{S-OL}_{n-1},\text{S-OL}_n \}$$

Each list, S-OL_x, is a list of ordered pairs (R_{xa},A_{xb}) as defined above. However, the first member of S-OL_x, S, represents the subject reference. The remaining ordered pairs within S-OL_x, OL, are object references particularly related to the subject reference in the first position.

The reader should be aware that an unlimited number of subjects can be associated with any given verb. Furthermore, an unlimited number of objects can be related to each subject. All of these relationships are built during an interactive session with the DBA. The internal representation is hidden from the DBA. The generated verb lists are a representation of the relationships in the underlying database. For a detailed example, see Appendix F.

Using the verbs, nouns representing the object and subject can now be examined to isolate the qualifier field. The logic requiring the correct identification of the subject and object during the parse will now become apparent. The target field may be either the subject or the object. The general process of finding the qualifier field requires searching the lists associated with the identified verb (remember S-OL_x?) matching an ordered pair reference of the type (R_{xa}, A_{xb}) with the ordered pair given by the target attribute (R_m,A_n). If the target attribute is identified as a subject reference, the pair R_m,A_n) is compared to each S of S-OL_x of the verb list until a match is found. That is, the subject reference of the target attribute is compared with each subject reference of the verb list until either a match is found, or both the target and the verb list have been completely searched. If, however, the target attribute has been identified as an object, the pair (R_m,A_n) is compared to each (R_{xa}, A_{xb}) found in OL of each S-OL_x list. In this case, the object reference of the target attribute is being compared to the list of objects associated with each subject. In either case, if a match is made, the corresponding subject or object field associated with the matched target field is identified as the qualifier field of the query.

Three possibilities exist during the process of searching and matching the various lists. First, a match may occur, which progresses as above. Second, no match may occur, in which case the qualifier field is not identifiable. This could be the result of one of two problems. If the relationships are not properly defined logically by the DBA, the end user will be affected during his session. If a detailed query was given, and the processor returns general results, a poorly defined schema representation is suspected, a DBA error. If the relationships are well defined, a user error is possible, indicating he cannot identify the domain of the database. Generally, this is exposed earlier in the analysis with the user struggling to provide synonyms for every word in his query.

The second possibility exists if there are conditions in which the user may request a general return of information such as:

USER: List all customers.

Such requests may eliminate the need for a qualifier field. Flags in the data structure may suggest the occurrence of this condition during a parse, by noting the use of the certain key words such as "list" combined with the successful recognition of sentences in particular parse rules. No qualifier field is isolated, and a more general formal query is a reasonable result.

The third possibility is the occurrence of multiple matches. Multiple matches are allowed, but are not widespread. The NBASE system, currently, takes the first match and attempts to resolve it. All other matches are thrown away. Early enhancements may include adding a semantic resolver to identify the proper relationship. Priority can be assigned during a preprocessing session to the stronger relationship, NBASE then guaranteeing the priority relationship to be processed if multiple matches exist.

Once the target attribute and the primary qualifier attribute are identified, a formal query can be partially generated in the form (using MISTRESS commands) of

SELECT target-attribute FROM table WHERE qualifier attribute = ' '.

The 'table' is identified as the relation associated with the attribute in the target parse.

Before examing the process by which the command is further expanded, recall that there are query formats that do not yield a target attribute during the parse. The existing formats are the 'who' and the 'where' interrogatives. Introduced in chapter 3 were special lists, maintained during preprocessing for the identification of 'who' and 'where' attributes. A direct identification of a target attribute cannot be made when such a query is presented. This condition is flagged during the parse, signaling the semantic analyzer to resolve the reference. The parse, though unable to identify the target attribute, must indicate the position of the target attribute as an object or a subject. The process of isolating the proper target attribute is similar to that of a qualifier attribute, using the list of "whos" or "wheres" to match against the list available from the verb. If a target attribute is not available, an error condition exists, and no formal query is issued to the database. An error may be generated

for reasons similar to those described above.

The next function completes the shortest query allowed using qualifier attributes. Associated with each qualifier, is a value that the database would be searched for. Using the example:

USER: Where does Julie Richardson live?

the string 'Julie Richardson' is a qualifier value associated with some qualifier attribute that has been identified using the 'where' list and the lists associated with 'live'. Qualifier values allow for the return of specific information related to the qualifier value being searched for. Appending the qualifier value to the formal command is trivial in the NBASE system. The actual values associated with the subject or object noun(s) are passed to the semantic analyzer. These values, often volatile, generally represent the actual values to be used, and are appended to the formal query. Hooks have been provided in other components of the system allowing for more complex processing in which such values would not be proper qualifier values. Currently, they are not fully implemented, therefore, additional analysis is not required. Another resolver will be necessary if the hooks are implemented and more than a direct attribute value is generated.

In a larger database, a more refined query is desired which is more detailed than

SELECT STREET FROM ADDRESS WHERE f# = 'Julie Richardson.'

Several fields may be examined to further delimit the amount of data returned to the user. For example

SELECT STREET FROM fADDRESS WHERE f# = 'Julie Richardson'
AND STATE = 'MICHIGAN';

A corresponding natural language query might be:

What street does Julie Richardson live on, in Michigan?

'Julie Richardson' is the primary subject reference, whose qualifier attribute is isolated as described above. 'Michigan' represents a secondary reference, usually associated with additional prepositional phrases. It is required that extra qualifier fields be identifiable to expand

the formal query to the user's expectation.

Like verbs, prepositions are associated with specified fields during the preprocessing session. The result provides further structure in the knowledge base, linking subjects and objects together through the preposition. The heuristic implemented to resolve secondary references does work, although weakly. Generally, more associations can be made through preposition than through a particular verb. One preposition, then, may have an expanded list of possible matches. As long as duplicates cannot be processed and later reduced to one, the NBASE secondary qualifier analysis represents a weak link in the system that requires the user to carefully think out his question in order to increase the probability of recovering the proper information. The system will return correct information, possibly isolating a field other than the one requested by the user, choosing the first relation-attribute pair matched from which to generate a query.

The additional processing required to handle duplicates in subject-verb-object relations is postponed for future enhancements.

7. THE PROLOG APPROACH

7.1. PROLOG Overview

The PROLOG programming language was developed early in the 1970's and focuses on solving problems that involve objects and the relationships between objects. Introduced as a language for PROgramming in LOGic, PROLOG is being used for understanding natural language and other areas of artificial intelligence research (Clocksin, 1981).

PROLOG was slow to gain acceptance, being overshadowed both by other programming languages (e.g. FORTRAN) and a conservative approach to problem solving (Von-Neumann style). As research revealed that programming language structure could be efficiently specified by the supporting architecture of the computer, symbolic languages (e.g. PROLOG, LISP, SNOBOL) became recognized as valuable tools, particularly in AI (Backus, 1978). Such languages were found to be totally provable without test runs or simulation. Designed to make problem solving logical, symbolic language programming typically results in large reductions in code, easy module modification, and much less debugging. As a symbolic programming language, PROLOG is now recognized as a practical and efficient implementation tool for intelligent program execution (Clocksin, 1981).

Specifying an algorithm using a logic based programming language contrasts with the methods employed by using traditional languages. Those who have 'grown up' on the mathematical flavor of procedural logic often find it difficult to convert to a symbolic approach. Algorithm development centers around the manipulation of symbols rather than specifying a procedural process. It appears that the strongest argument against symbolic pro-

gramming is the necessary transition to get from seeing how such a language works, to where one has to be in order to program (Winston, 1984). It is well worth the time investment!

In PROLOG, processing actions are described by isolating the existence of all objects and defining the formal relationships among those objects. These represent the facts of the logical system being described. After definitions are made, solutions are derived according to the "truthfulness" of the existing state of objects. In PROLOG, such decision processing is not limited to the Boolean operations of "AND", "OR", and "NOT", but includes extensions into set theory which make PROLOG quite different and very powerful (cf. Li, 1984). PROLOG program execution is defined by any one, or combination of, the following PROLOG controls:

1. The logical declaratives of PROLOG.
2. Inferences made from previously resolved facts.
3. Explicit control information provided by the programmer.

The above programming approach, emphasizes descriptions of known facts and relationships about a problem (points 1 & 2), rather than listing the sequence of steps necessary to solve the problem (point 3).

Using a simple example about family relationships, we demonstrate the logical orientation of PROLOG.

> "Mikel loves all those in the Brown family. If Julie, Sarah and James are members of the Brown family, then we can infer that Mikel loves Julie, Sarah and James."

In PROLOG, these facts and inference rules are expressed as follows:

1. loves(mikel, brown) - Mikel loves those in the Brown family.
2. member(julie,brown) - Julie is a member of the Brown family.
3. member(sarah, brown) - Sarah is a member of the Brown family.
4. member(james,brown) - James is a member of the Brown family.

5. loves(X,Y,Z) :- loves(X,Z), member(Y,Z).

Rule 5 states: Subject X loves object Y, if X loves the Z family and Y is a member of the Z family. OR ...

Mikel loves [Julie|Sarah|James] if Mikel loves the Browns and [Julie|Sarah|James] is a member of the Browns.

Several programming conveniences are worth noting here. First, this is a *complete* PROLOG program, able to respond to a user's request by examining the truthfulness of the incoming statement concerning certain family relationships. Second, statements 1-4 represent PROLOG declarative facts; statement 5 is an inference rule. Given a set of facts, note the generality of the inference rule with the use of variables X,Y,Z, allowing for the evaluation of any (family,member) relationship. Third, is the interpretation applied to the PROLOG definitions asserted. Meaning (or semantics) is controlled by the designer. One construct such as "loves(mikel,brown)" may represent different meanings to different programmers. This is a stark contrast to traditional programming languages, where most elementary constructs are defined by the language specification. The answer to the common question "How does 'loves(mikel,brown)' define 'Mikel loves the Browns" is "Because the designer chose to apply that meaning to that particular construct." PROLOG structures mean what the programmer wants them to mean, and requires only a consistent definition throughout program execution.

There are two ways in which PROLOG semantics are defined, procedural and declarative. Procedural semantics are the typical state descriptions that exist at any particular cycle during program execution. The procedural description identifies the way a goal is executed. Declarative semantics are the set of descriptive statements about the program (e.g Mikel loves the Browns). Generally, such declarations are represented as n-ary statements in prefix form. (Using declaration 1. above, loves is the prefix operator; mikel and brown are the operands in this binary statement.) It is the combination of both semantic representations that make PROLOG a valuable tool for developing natural language systems.

In addition to PROLOG's deductive capacity, dual semantic interpretation, and specification-programming language, several other features are provided.

Like most symbolic languages, PROLOG supports recursion, while dispensing with loops (though they are indirectly implementable), gotos and assignment statements. Those developing natural language grammars have the capability to build ATN's which require recursion (cf. Wilcox, 1983). Additionally, PROLOG supports a grammar specification

language subset, that allows for direct definition of the grammar rules within the program (Appendix G).

Procedures may have multiple outputs as well as multiple inputs. Beyond providing for multiple parameters, PROLOG allows a procedure to deduce, or make several inferences on the same base of data. Generally, only one inference is passed on. PROLOG supports non-determinism, however, allowing multiple alternatives in the processing path to be examined. Built in to PROLOG, is a backtracking capability that automatically is invoked whenever an earlier alternative fails to successfully complete its path. Such failures can be forced, constraining the backtracking mechanism to work like a looping structure similar to a repeat-until loop. Backtracking can be turned off using the PROLOG "cut" symbol (!). The cut symbol stops backtracking from backing up beyond the location of the cut. The support of non-deterministic processing is worth much to natural language research. Ambiguity, being a primary natural language concern, can be directly addressed by resolving all ambiguous paths (non-determinism) and applying various heuristics to select the best alternative from all outcomes.

PROLOG's primary computation mechanism is based on pattern matching, including a pattern directed procedure call. Procedures are successfully invoked if the actual parameter patterns match the formal parameter patterns. PROLOG provides an "uninstantiated" variable, an uninitialized pattern variable, that will match the first pattern it is compared with, while assuming its value. This method of variable instantiation is the fundamental assignment construct in PROLOG.

Like Lisp, PROLOG provides a uniform data structure, called the term, from which both the program and the data are formed. A term is defined recursively as follows:

1. A constant is a term.
2. A variable is a term.
3. If f is a function symbol, and $X_1,...,X_n$ are terms
   then $f(X_1,...,X_n)$ is a term.
4. All terms are generated from the above rules.

The uniform data structure is important to AI applications, allowing a program to modify

itself, when necessary, to complete a task. There are obvious cautions to be introduced with such program modifications.

The generality of the PROLOG record structure allows for an unlimited number of record types and an unlimited number of record fields with no type restrictions. Nested records are allowable, and a specification mechanism is provided for list manipulation.

The database nature of PROLOG permits new facts to be asserted to the database during runtime. Using the built-in functions of assert and retract, a PROLOG program has learning capabilities, asserting facts where inferences have produced a degree of truthfulness. These features allow new facts to be added, outdated data to be released, and, when combined with several other built-in functions, they provide a powerful mechanism for program modification. Furthermore, the PROLOG interpreter interfaces with the underlying operating system, providing file manipulation capabilities and a user callable "system" call. File i/o is instrumental for designing a system with learning functions that are permanent (i.e. are maintained from execution to execution) without saving an explicit copy of the run time module.

7.2. PROLOG, Relational Databases and Logic

Without presenting an indepth treatment of database theory, the following compares the underlying structure of PROLOG to relational database design. Comparisons will reveal both PROLOG's strength for database implementation and a weakness of natural language front end processors.

Databases are systems of data control that focus on maintaining data independence (the ability to change storage or access strategies without modifying the application), and limiting data redundancy (multiple representation of data, resulting in inconsistencies and wasted resources). A relational database provides a high degree of logical data manipulation while maintaining data independence and integrity (cf. Date, 1983). The simplicity and generality of the relational model make it a powerful organization, but not necessarily an easy one to implement.

A relational database is organized using tables of attributes as its primary construct (Appendix C). By imposing a relational semantic structure over a PROLOG declaration, PROLOG becomes a natural medium for database representation[1]. The following PROLOG declaration

relation1(attribute1, attribute2, ..., attributeX).

may be used to define a relational table, relation1, whose fields are attribute1 through attributeX. This declaration is simply extended and supported by additional declaratives to complete some set, A, of relational structures. If set A, represents the structural design of some database, B, the entire relational data structure of B is simply defined using PROLOG.

The power of a relational database lies in its ability to perform logical operations over its data structure to produce requested information. The set of logical operators include the simpler operations of intersection (AND), union (OR), negation (NOT), set difference and Cartesian Product. The relational approach is additionally supported by the more powerful relational (logical) operations of selection, join, projection and division (Date, 1983). The operators are used for isolating columns of various relations, joining them and delimiting the targeted information.

A data base management system (DBMS) can be described logically as a general purpose, question-answering system. The set of facts that drive the system can be thought of as axioms of a theorem, where the questions asked of the database stands for the conclusion of the theorem. Since logic is the study of implication between assumptions and conclusions, we note the closeness associated with a relational DBMS and logic programming (Dahl, 1982). Having introduced PROLOG as a logical programming language, with extended set operations, PROLOG becomes a leading consideration for complete database implementation.

While the implementation of a logic programmed database requires addressing several optimization issues for query resolution (cf. Li, 1984), once constructed, it does allow a more efficient interface with a natural language front end processor. The advantage exists in the

[1]The PROLOG interpreter considers all PROLOG facts and assertions to be a not necessarilily relational database of information.

property that both the database and query processor are logic programmed. One of the foremost shortcomings of a natural language front end processor is the necessity for manipulating data in "natural language form," only to store it using an alternate organizational structure. Fundamentally structured natural language data should be left as such, not crammed into a format that does not represent it well . Data that is processed across differing structures is often stored in a way that renders it hard to get (Woods, 1977). Such a representation demands more from the user, conflicting with the purpose of designing a natural language system, which is to provide complete access to information in a trivial, straight foreword way.

The state of the art is to interface a natural language front end processor with a more formally implemented database. The natural language query requires a transformation to a formal language query, with which the database is interrogated. An alternate approach is the logic programmed database. The user query is in a natural language format, requiring no transformation before accessing the database, also in natural language format. Not only is the efficiency increased by reducing transformations, but access to data is increased through direct representation of natural language.

Currently, there are no marketed logic programmed systems available. Some models do exist, and many prototypes exist that have been implemented for research. The NBASE system is a front-end processor, interfaced with a formal query database, called Mistress (Rhodnius, 1983). NBASE is programmed in PROLOG. By combining the knowledge available for natural language processing, logic programming, relational database theory, and tools such as PROLOG, the immediate future should be prosperous for research and development of logic programmed database systems.

8. CONCLUSIONS

Natural language systems have grown beyond model and prototype products. Several large systems have been generated with varying degrees of success. INTELLECT has entered the commercial market and claims a 90% success rate for correctly answering user queries (Davis, 1983). Additional research has advanced understanding of natural language processing to a place where competitive products are emerging, giving the inexperienced user access to large amounts of electronically stored information (Ballard, 1982; Biermann, 1982; Kaplan, 1984).

Solving all the problems inherent in a natural language fact retrieval system, however, is still a long way off (Kolodner, 1983; Schneidermann, 1981; Smith, 1980). The assumption that ordinary English is an ideal way to communicate with machines is not well founded. Representing the complete set of English sentences is an unending task that cannot be defined by the limited resources of a computer. Efforts such as Biermann's (1982) to generate a fortual language subset scope the project down to a manageable size but restricts the user. Avoiding semantic overshoot (request of non-represented information) is still best accomplished by educating the user in the domain of the knowledge base, with the excuse that some understanding of the system is important.

Representing the knowledge base is the first major concern of any natural language processor. The set of knowledge constructs provided, directly determines the range of questions available to the user. The process of knowledge acquisition is important for portability and flexibility considerations. Preprocessors such as PREP (Ballard, 1984) and JPREP, which interface with the evaluation module, yield increased portability by offering a method of

knowledge base construction transportable to any domain.

The difficulty of semantic analysis and parsing a natural language typifies the design considerations required for natural language translation. Ambiguity cannot be ignored. Instead, methods must be employed to resolve expected ambiguous statements, often requiring user interaction. A successful system will converse with the user, when necessary, hiding all technical details. The ideal system supplies a degree of context sensitivity to resolve vague statements expressed by a user. Opening doors to inexperienced users by exploiting their knowledge of a natural language, requires fully emulating the thinking process of man, a process that has only begun to be understood. Ambiguity and vagueness, though understood, are two obstacles that require much design effort during the production of a natural language system.

Has computer science effectively humanized computer information systems? Has AI, using natural language, made information systems more accessible to a more general class of users? Currently, natural language systems have been found appropriate for those users who have much understanding of the semantic knowledge in the system being used, but are unfamiliar with the formal system syntax for achieving the goals. Some would argue that this is insufficient; the goal is to supply access to all users. Others argue that those using such a system would likely have the needed domain understanding by virtue of having a specific information need. Despite one's philosophy, recent advances have still left severe limitations on what can be provided. The immediate challenge lies in the necessary increase of understanding required in the fields of logic, linguistics and computer science, before automated natural language comprehension will pass what is still considered to be "elementary stages" (Kelly, 1984; Smith, 1980).

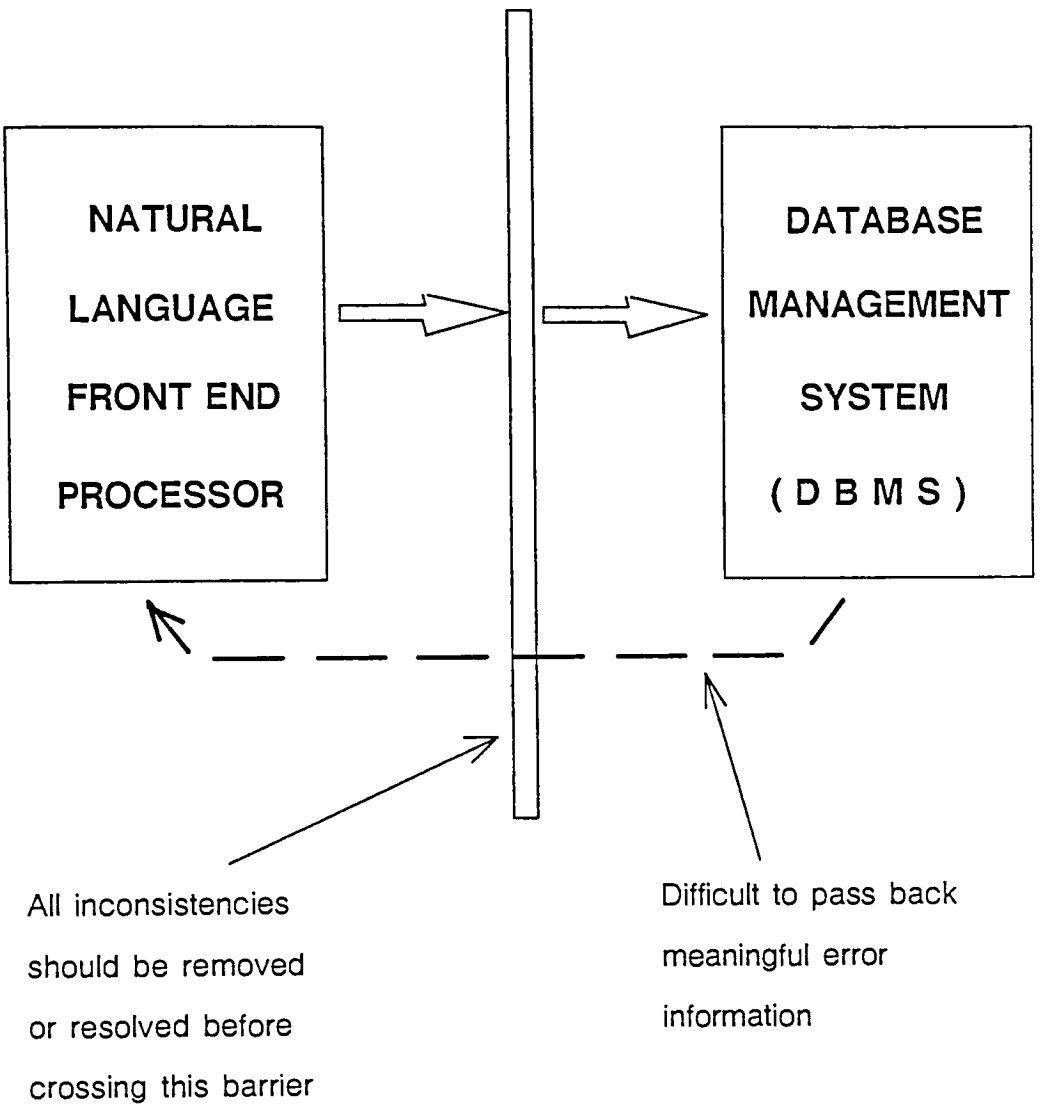
The NBASE system is a small prototype natural language front-end processor to a formal database, MISTRESS (Rhodnius, 1983). The purpose of the NBASE design was not to provide a commercial product, but to initiate research in natural language processing. The NBASE system is portable, providing a preprocessor, JPREP, for knowledge base construc-

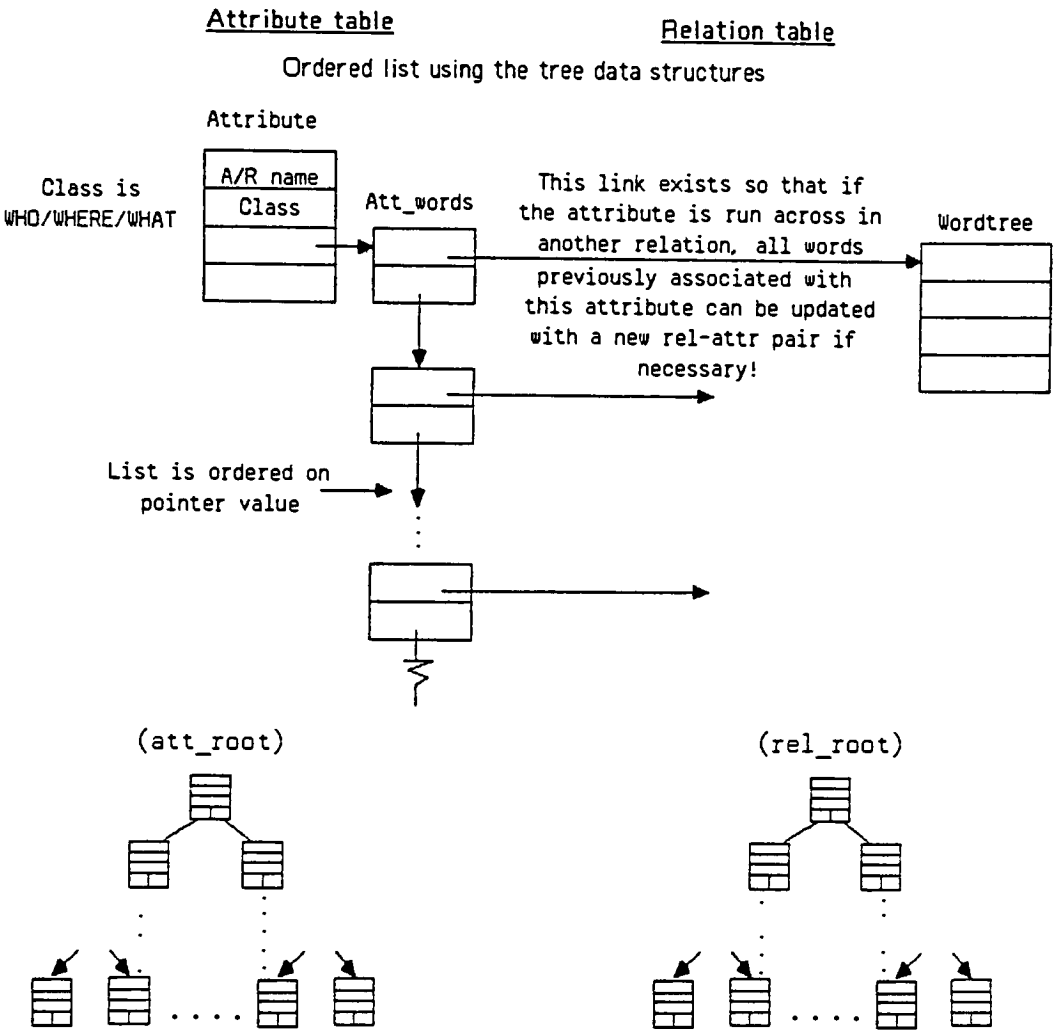
tion, which has been displayed to be domain independent. Successful preprocessing does require a complete understanding of database schema and content, and is usually performed by the DBA.

Ambiguities were addressed, but not exhaustively resolved. As previously mentioned, vagueness heuristics were not employed, leaving room for system enhancement by others. The grammar would be better implemented using an ATN, an enhancement that would be a nice project for an AI course. Furthermore, changes in the grammar are necessary to better isolate error conditions and to extend the fortual language currently supplied by the system.

A spelling checker would be another feature that would upgrade the current system, and some hooks for that have been provided. NBASE is only functioning as a query system and does not provide for creating and updating the database. Again some hooks for this have been provided. User extension of the grammar during run time, employing a meta language and production rules would also be an interesting feature to add to NBASE. Each of the above would supply challenging individual topics and projects for a one term AI course.

Finally, the design and implementation of a complete natural language database system would provide a challenging thesis topic for those interested in both database implementation and AI. Using a logic based language like PROLOG, many of the previously mentioned weaknesses of a front-end processor could be eliminated by such a system.


NATURAL
LANGUAGE
FRONT END
PROCESSOR
DATABASE
MANAGEMENT
SYSTEM
(D B M S)
All inconsistencies
should be removed
or resolved before
crossing this barrier
Difficult to pass back
meaningful error
information



(curatt) --> current attribute node
(currel) --> current relation node

Functions

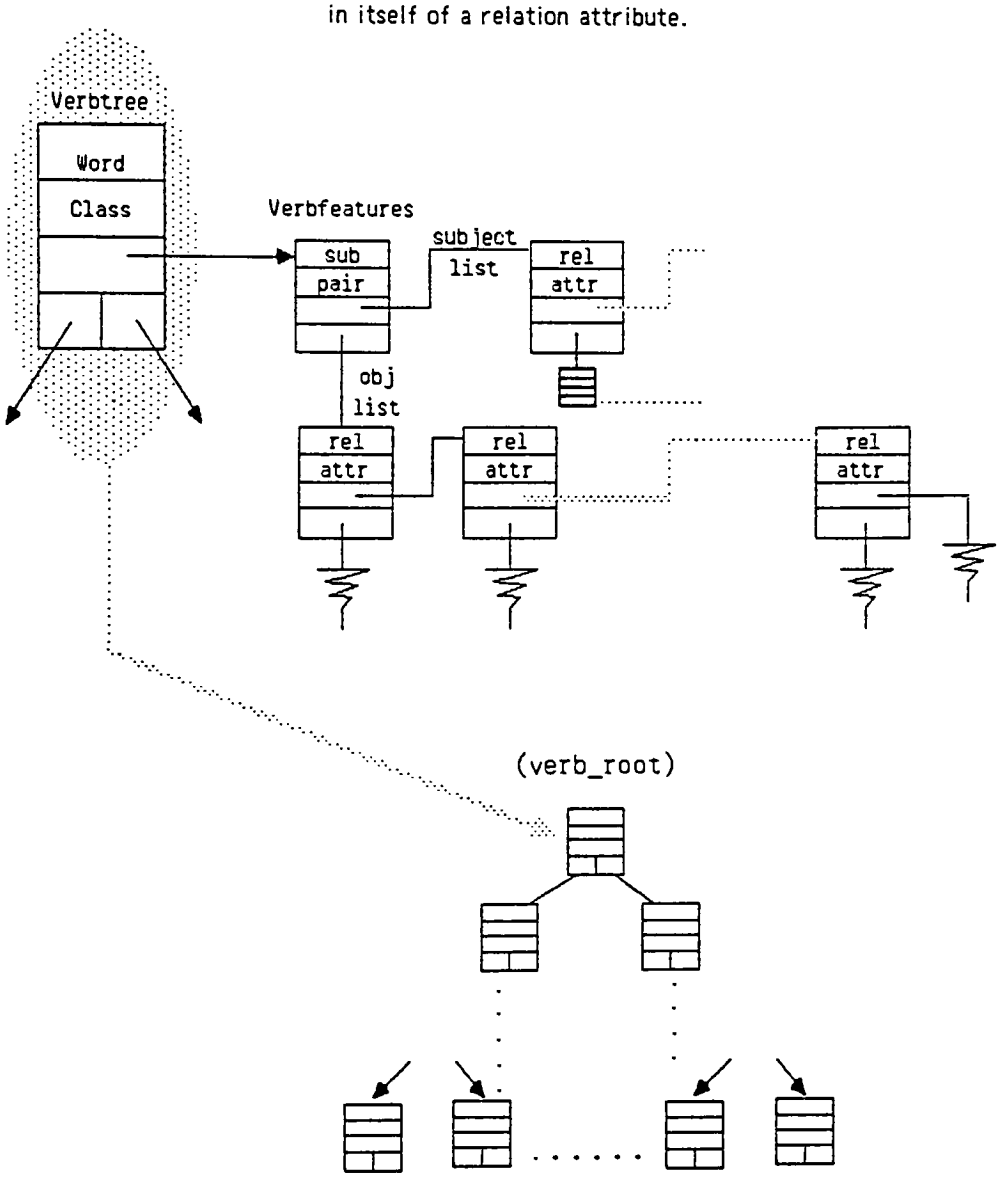
insertval (name of rel/attr, current rel/attr pointer, root of
proper rel/attr tree)
if the rel/attr does not exist then load it into tree
else set a flag for special action later

set_att (root node of current word)
if there is not a list of words or new word is good for a first
entry then install new word at the begining of the list
else check next word in the list

install new root word (which is a pointer) if necessary

Verb—preposition Table

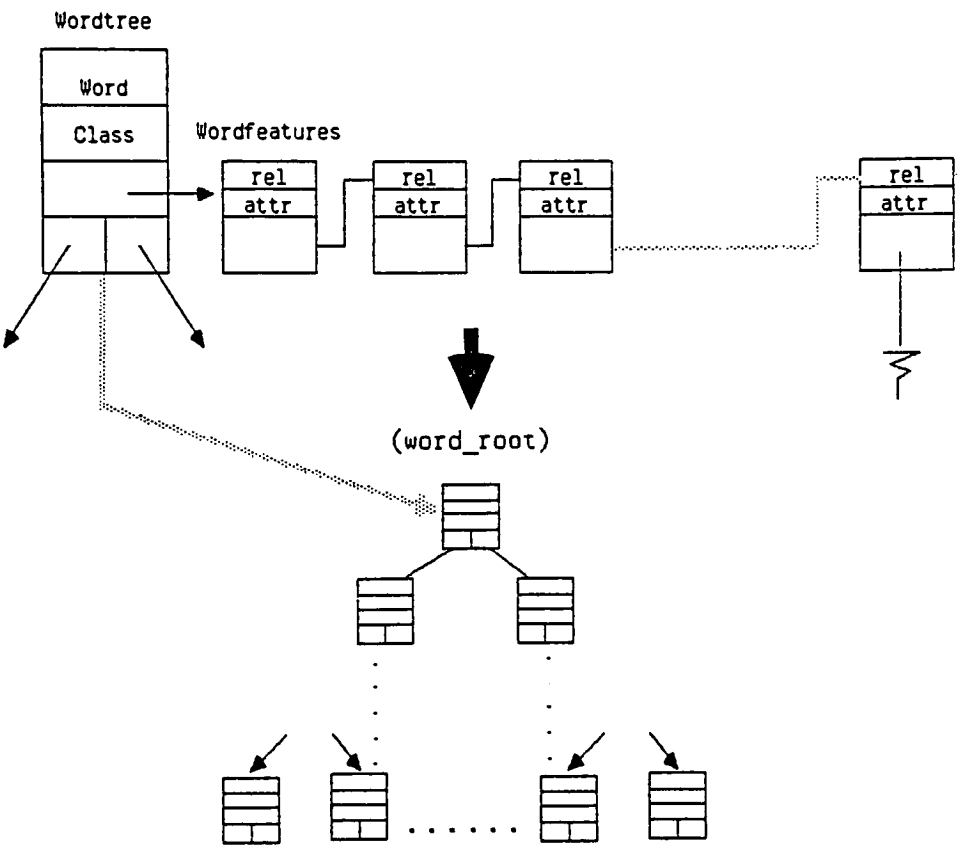
Associated with every verb lists of pairs of pairs, consisting of an object—subject pair, each object/subject being a pair. in itself of a relation attribute.



Tree d.s.

Noun–Adjective Table

Associated with every word is a (relation–attribute) list
representing the attributes with which the word is associated.



Functions :

```
-insertword ( Pointer to name of word, root of tree, class of word)
              if word is not recorded, record it
              set the features list
              set the attribute list
-setfeature ( feature list pointer of current node )
              if current rel/attr not associated with this word,
                  add it to the list
```

Example Relations

faddress

f#	street	city	state	zip

fstate

f#	cnum	church

```
prep thesis

Re/Initialize current environment ? (y/n): y
Please list the Primary Key field(s) separated by blanks
fields --> L#

**********

Is the attribute 'Artist' a
        A. Person
        B. Place
        C. Thing
        D. None of the Above?!
attribute --> A

List the synonyms for 'Artist' (one/line; <return> to quit):
Synonym --> artist
Synonym --> musician
Synonym --> instrumentalist
Synonym -->
List all adjectives that could refer to 'Artist'  (one/line; <cr> to quit):
Word -->
List all verbs that could use 'Artist' as a subject  (one/line; <cr> to quit):
Word --> plays
List all relation-attribute pairs which could be objects on
        which the verb 'plays' is manipulated by the subject 'Artist'

        One pair/line (e.g.) <relation> <attribute>
Pair --> GIGS Location
Pair --> INSTRUMENT Instrument
Pair -->

Word --> recorded
List all relation-attribute pairs which could be objects on
        which the verb 'recorded' is manipulated by the subject 'Artist'

        One pair/line (e.g.) <relation> <attribute>
Pair --> RECORD Title
Pair -->

Word -->
List all prepositions that manipulate with 'Artist'  (one/line; <cr> to quit):
Word -->

Is the attribute 'Title' a
        A. Person
        B. Place
        C. Thing
        D. None of the Above?!
attribute --> C
        .
        .
        .
```

What now ? What album name did Ted Sanquist record?

1. analyze([What,album,name,did,Ted,Sanquist,record,?], _65635)
2. analyze([What,album,name,did,Ted,Sanquist,record,?],
   [w_token(what,q_word,_493),w_token(album,noun,[[RECORD,Title]]),
   w_token(name,noun,[[RECORD,Title]]),w_token(did,aux_verb,_558),
   w_token(Ted,noun,[volatile]),w_token(Sanquist,noun,[volatile]),
   w_token(record,verb,[[[RECORD,Artist],[RECORD,Title]]]),p_token(?)])

1. parse(_6563,[w_token(what,q_word,_493),w_token(album,noun,[[RECORD,Title]]),
   w_token(name,noun,[[RECORD,Title]]),w_token(did,aux_verb,_558),
   w_token(Ted,noun,[volatile]),w_token(Sanquist,noun,[volatile]),
   w_token(record,verb,[[[RECORD,Artist],[RECORD,Title]]]),p_token(?)],[])
2. parse(ptree(type(q1),qv([Ted,Sanquist],[volatile,volatile]),
   ta([[[RECORD],Title]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null)),
   [w_token(what,q_word,_493),w_token(album,noun,[[RECORD,Title]]),
   w_token(name,noun,[[RECORD,Title]]),w_token(did,aux_verb,_558),
   w_token(Ted,noun,[volatile]),w_token(Sanquist,noun,[volatile]),
   w_token(record,verb,[[[RECORD,Artist],[RECORD,Title]]]),p_token(?)],[])

1. semantics(ptree(type(q1),qv([Ted,Sanquist],[volatile,volatile]),
   ta([[[RECORD],Title]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null)))
2. semantics(ptree(type(q1),qv([Ted,Sanquist],[volatile,volatile]),
   ta([[[RECORD],Title]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null)))

mscmd thesis "select Title from RECORD where RECORD.Artist = 'Ted Sanquist'"
Title

Courts of the KING

What now ? What artist recorded Unto Him?

1. analyze([What,artist,recorded,Unto,Him,?],_65635)
2. analyze([What,artist,recorded,Unto,Him,?],[w_token(what,q_word,_401),
   w_token(artist,noun,[[RECORD,Artist]]),
   w_token(recorded,verb,[[[RECORD,Artist],[RECORD,Title]]]),
   w_token(Unto,noun,[volatile]),w_token(Him,noun,[volatile]),p_token(?)])

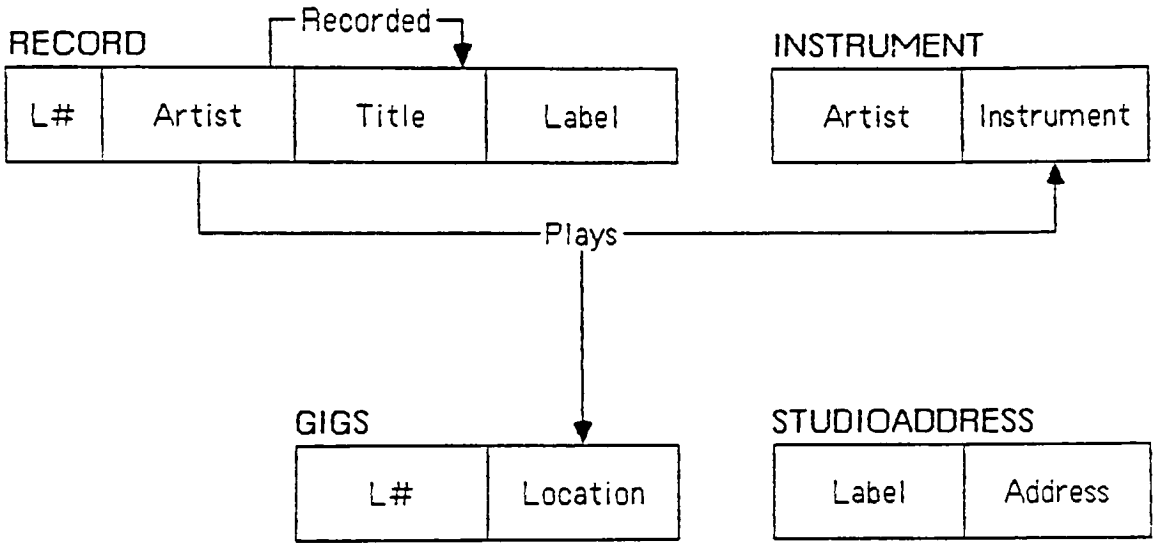
1. parse(_65636,[w_token(what,q_word,_401),
   w_token(artist,noun,[[RECORD,Artist]]),
   w_token(recorded,verb,[[[RECORD,Artist],[RECORD,Title]]]),
   w_token(Unto,noun,[volatile]),w_token(Him,noun,[volatile]),p_token(?)]).
2. parse(ptree(type(q5),qv([Unto,Him],[volatile,volatile]),
   ta([[[RECORD],Artist]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null),
   [w_token(what,q_word,_401),w_token(artist,noun,[[RECORD,Artist]]),
   w_token(recorded,verb,[[[RECORD,Artist],[RECORD,Title]]]),
   w_token(Unto,noun,[volatile]),w_token(Him,noun,[volatile]),p_token(?)]).

1. semantics(ptree(type(q5),qv([Unto,Him],[volatile,volatile]),
   ta([[[RECORD],Artist]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null))
2. semantics(ptree(type(q5),qv([Unto,Him],[volatile,volatile]),
   ta([[[RECORD],Artist]]),vl([[[RECORD,Artist],[RECORD,Title]]]),pl(null))

mscmd thesis "select Artist from RECORD where RECORD.Title = 'Unto Him'"
Artist

The Bridge

The first entry = input data and its data structure.
The second entry = output data and its data structure.



Plays:
{{(RECORD,Artist),(INSTRUMENT,Instrument),(GIGS,Location)}}
Records:
{{(RECORD,Artist),(RECORD,Title)}}

Prolog:
(plays,[[[RECORD,Artists),[INSTRUMENT,Instrument],[GIGS,Location]]]).
(RECORDS,[[[RECORD,Artists],[RECORD,Title]]]).

```
/*              grammar
**
**      This file defines the grammar used for the natural language
** processor.
*/

parse(ptree(Type,Qual,Targ,Verb,Prep))          -->
                interogative(Type,Qual,Targ,Verb,Prep),pn,{!}.
parse(ptree(Qual,Verb,Targ))            -->
                declaration(Qual,Verb,Targ),pn,{!}.
declaration(S,V,O) --> noun_phrase(S),verb_phrase(V,O), {!}.
declaration(S,_,O) --> noun_phrase(O),{!}.

/*1*/

interogative(type(q1),qv(S,F),ta(O),vl(V),pl(P))          -->
                qwd(what),olist(O),aux(V1),slist(S,F),vlist(V),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(null))       -->
                qwd(what),olist(O),aux(V1),slist(S,F),vlist(V),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(P))          -->
                qwd(what),vlist(V),olist(O),plist(P),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(null),ta(O),vl(V),pl(P))         -->
                qwd(what),vlist(V),olist(O),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(P))          -->
                qwd(what),aux(A),olist(O),vlist(V),agnt(by),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(null))       -->
                qwd(what),aux(A),olist(O),vlist(V),agnt(by),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(P)) -->
                qwd(what),olist(O),vlist(V),agnt(by),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q1),qv(S,F),ta(O),vl(V),pl(null))       -->
                qwd(what),olist(O),vlist(V),agnt(by),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q5),qv(S,F),ta(O),vl(V),pl(P))          -->
                qwd(what),aux(A),olist(O),agnt(who),vlist(V),slist(S,F),
                plist(P), {!,loadregister(subject,target)}.
interogative(type(q5),qv(S,F),ta(O),vl(V),pl(null))       -->
                qwd(what),aux(A),olist(O),agnt(who),vlist(V),slist(S,F),
                {!,loadregister(subject,target)}.
interogative(type(q5),qv(S,F),ta(O),vl(V),pl(P)) -->
                qwd(what),olist(O),vlist(V),slist(S,F),plist(P),
                {!,loadregister(subject,target)}.
interogative(type(q5),qv(S,F),ta(O),vl(V),pl(null))       -->
                qwd(what),olist(O),vlist(V),slist(S,F),
                {!,loadregister(subject,target)}.
interogative(type(q5),qv(null),ta(O),vl(V),pl(P))         -->
                qwd(what),olist(O),vlist(V),plist(P),
```

```
                {!,loadregister(subject,target)}.
interogative(type(q10),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(who),vlist(V),slist(S,F),plist(P),
                {!,loadregister(subject,target)}.
interogative(type(q10),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(who),vlist(V),slist(S,F),{!,loadregister(subject,target)}.
interogative(type(q10),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(who),vlist(V),agnt(by),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q10),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(who),vlist(V),agnt(by),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q20),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(how),aux(A),slist(S,F),vlist(V),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q20),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(how),aux(A),slist(S,F),vlist(V),
                {!,loadregister(object,target)}.
interogative(type(q20),qv(S,F),ta(null),vl(null),pl(P))   -->
                qwd(how),aux(A),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q20),qv(S,F),ta(null),vl(null),pl(null)) -->
                qwd(how),aux(A),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q30),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(when),aux(A),slist(S,F),vlist(V),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q30),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(when),aux(A),slist(S,F),vlist(V),
                {!,loadregister(object,target)}.
interogative(type(q30),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(when),aux(A),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q30),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(when),aux(A),slist(S,F),
                {!,loadregister(object,target)}.
interogative(type(q40),qv(S,F),ta(null),vl(V),pl(P))      -->
                qwd(where),aux(A),slist(S,F),vlist(V),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q40),qv(S,F),ta(null),vl(V),pl(null))   -->
                qwd(where),aux(A),slist(S,F),vlist(V),
                {!,loadregister(object,target)}.
interogative(type(q40),qv(S,F),ta(null),vl(null),pl(P))   -->
                qwd(where),aux(A),slist(S,F),plist(P),
                {!,loadregister(object,target)}.
interogative(type(q40),qv(S,F),ta(null),vl(null),pl(null)) -->
                qwd(where),aux(A),slist(S,F),
                {!,loadregister(object,target)}.

olist(List)     -->
                target_att(Targ),conj,olist(More),{!,append([Targ],More,List)}.
olist([List])   -->
                target_att(List),{!}.
```

```
/*                    knowledge
**
**      This file is the knowledge base for the evaluation of
** questions given to the natural language processing system. The
** knowledge base is provided as a start base only. The system does
** have the capability to learn. The additional learned features
** may or may not be locatable in this file.
**
*/

is_a(noun,speech_part).
is_a(verb,speech_part).
is_a(aux_verb,speech_part).
is_a(determiner,speech_part).
is_a(q_word,speech_part).
is_a(conjunctive,speech_part).
is_a(preposition,speech_part).


is_a(the,determiner,_,_).
is_a(an,determiner,_,_).
is_a(a,determiner,_,_).

is_a(is,aux_verb,_,_).
is_a(has,aux_verb,is,_).
is_a(does,aux_verb,_,_).
is_a(did,aux_verb,does,_).

is_a(who,q_word,_,_).
is_a(how,q_word,_,_).
is_a(what,q_word,_,_).
is_a(when,q_word,_,_).
is_a(where,q_word,_,_).

is_a(by,agword,_,_).
is_a(who,agword,_,_).
is_a(that,agword,_,_).

is_a(and,conjunctive,_,_).
is_a(',',conjunctive,_,_).

is_a('GIGS',noun,'GIGS',[['GIGS','GIGS']]).
is_a('INSTRUMENT',noun,'INSTRUMENT',
                [['INSTRUMENT','INSTRUMENT']]).
is_a('RECORD',noun,'RECORD',[['RECORD','RECORD']]).
is_a('STUDIOADDRESS',noun,'STUDIOADDRESS',[['STUDIOADDRESS','STUDIOADDRESS']]).
is_a('Address',noun,'Address',[['STUDIOADDRESS','Address']]).
is_a('Artist',noun,'Artist',[['RECORD','Artist']]).
is_a('Instrument',noun,'Instrument',[['INSTRUMENT','Instrument']]).
is_a('L#',noun,'L#',[['GIGS','L#'],['INSTRUMENT','L#'],['RECORD','L#']]).
is_a('Label',noun,'Label',[['RECORD','Label'],['STUDIOADDRESS','Label']]).
is_a('Location',noun,'Location',[['GIGS','Location']]).
is_a('Title',noun,'Title',[['RECORD','Title']]).
```

```
is_a('address',noun,'address',[['STUDIOADDRESS','Address']]).
is_a('album',noun,'album',[['RECORD','Title']]).
is_a('artist',noun,'artist',[['RECORD','Artist']]).
is_a('instrument',noun,'instrument',[['INSTRUMENT','Instrument']]).
is_a('instrumentalist',noun,'instrumentalist',[['RECORD','Artist']]).
is_a('label',noun,'label',[['GIGS','L#'],['INSTRUMENT','L#'],['RECORD','L#'],
        ['RECORD','Label'],['STUDIOADDRESS','Label']]).
is_a('musician',noun,'musician',[['RECORD','Artist']]).
is_a('name',noun,'name',[['RECORD','Title']]).
is_a('number',noun,'number',[['GIGS','L#'],['INSTRUMENT','L#'],['RECORD','L#']]).
is_a('place',noun,'place',[['GIGS','Location']]).
is_a('plays',verb,'plays',[[['INSTRUMENT','Instrument'],['RECORD','Artist']],
        [['RECORD','Artist'],['GIGS','Location'],['INSTRUMENT','Instrument']]]).
is_a('record',verb,'record',[[['RECORD','Artist'],['RECORD','Title']]]).
is_a('recorded',verb,'recorded',[[['RECORD','Artist'],['RECORD','Title']]]).
is_a('title',noun,'title',[['RECORD','Title']]).
who(['Artist']).
where(['Address','Location']).


syn_attr(['label','number'],'L#').
syn_attr(['number'],'L#').
syn_attr(['place'],'Location').
syn_attr(['instrument'],'Instrument').
syn_attr(['artist'],'Artist').
syn_attr(['musician'],'Artist').
syn_attr(['instrumentalist'],'Artist').
syn_attr(['title'],'Title').
syn_attr(['album','name'],'Title').
syn_attr(['label'],'Label').
syn_attr(['address'],'Address').
```

BIBLIOGRAPHY

Aho, A. V. & J. D. Ullman (1978). *Principles of Compiler Design.* (Reading, Mass: Addison - Wesley Publishing Co., 1978).

Aho, A. V. & J. D. Ullman (1972). *The Theory of Parsing, Translation & Compiling, vol. 1 : Parsing.* (Englewood Cliffs : Prentice-Hall, Inc., 1972).

Backus, J. (1978). "Can Programming Be Liberated from the Von Neumann Style ? A Functional Style and Its Algebra of Programs." *Communications of the ACM,* vol. 21, no. 8. August, 1978.

Ballard, Bruce W. (1979). "Semantic and Procedural Processing For a Natural Language Programming System." *Tech. Rep. CS-1979-5.* Dept. of Computer Science, Duke University, Durham, N.C. May 1979.

Ballard, Bruce W. (1982). "A Domain Class Approach to Transportable Natural Language Processing." *Tech. Rep. CS-1982-11.* Dept. of Computer Science, Duke University, Durham, N.C. June 1982.

Ballard, Bruce W. & John C. Lusth (1982). "An English Language Processing Processing System Which Learns About New Domains." *Tech. Rep. CS-1982-18.* Dept. of Computer Science, Duke University, Durham, N.C. October 1982.

Ballard, Bruce W., John C. Lusth & Nancy L. Tinkham. (1984). "LDC-1: A Transportable, Knowledge-Based Natural Language Processor for Office Environments." *ACM Transactions on Office Information Systems.* vol. 2, no. 1. January, 1984.

Barr, Aaron & Edward A. Feigenbaum, eds. (1981). *The Handbook of Artificial Intelligence, vol. 1.* (Los Altos : William Kaufman, Inc., 1981). pp. 239-267.

Biermann, Alan W. (1982). "Natural Language Programming." *Tech. Rep. CS-1982-2.* Dept. of Computer Science, Duke University, Durham, N.C. January 1982.

Bobrow, D. G. & J.B. Fraser (1969). "An Augmented State Transition Network Analysis Procedure." *Proceedings of the International Joint Conference on Artificial Intelligence.* Washington D. C. (1969). pp. 557-567.

Chomsky, Noam & George Miller (1963). "Introduction to the Formal Analysis of Natural Languages." in Luce, R. D., Bush, R. R. & Galanter, E., eds. *Handbook of Mathematical Psychology.* (New York : John Wiley & Sons, 1963). pp. 269-321.

Chomsky, Noam. (1963). "Formal Properties of Grammars." in Luce, R. D., Bush, R. R. & Galanter, E., eds. *Handbook of Mathematical Psychology.* (New York : John Wiley & Sons, 1963). pp. 323-418.

Clocksin, William F. & Christopher S. Mellish (1981). *Programming in Prolog.* (New York: Springer-Verlag, 1981).

Dahl, Veronica (1982). "On Database Systems Development Through Logic." *ACM Transactions on Database Systems.* vol. 7, no. 1. March 1982. pp 102-123.

Date, C. J. (1983). *An Introduction to Database Systems.* (Reading, Mass: Addison - Wesley, 1983).

Date, C. J. (1981). *An Introduction to Database Systems.* (Reading, Mass: Addison - Wesley, 1981).

Davis, Dwight (1983). "Communicating With Computers in English: The Emergence of Natural Language Processing." *Mini-Micro Systems.* vol. 16, no. 11. October 1983.

Guida, Giovanni & Carlo Tasso (1983). "An Expert Intermediary System for Interactive Document Retrieval." *Automatica.* vol. 19, no. 6. November 1983. pp 759-766.

Harris, Larry R. (1977a). "ROBOT: A High Performance Natural Language Processor for Data Base Query." compiled by Waltz, David L. in "Natural Language Interfaces." *SIGART Newsletter.* no. 61. February 1977.

Harris, Larry. R. (1977b). "User Oriented Data Base Query with the ROBOT Natural Language Query System." *International Journal of Man-Machine Studies.* vol. 9, no. 6. November 1977.

Harris, Larry. R. (1978). "Experience with ROBOT in 12 Commercial Natural Language Data Base Query Applications." *Proceedings of the 6th International Joint Conference on Artificial Intelligence.* August 20-23, 1978. pp. 365-368.

Harris, Larry. R. (1984). "Natural Language Front Ends." in Winston, Patrick and Karen Prendergast, eds. *The AI Business: Commercial Uses of Artificial Intelligence.* (MIT Press, 1984).

Hayes-Roth, Frederick, Donald A. Waterman & Douglas B. Lenat, eds. (1983). *Building Expert Systems.* (Reading, Mass.: Addison-Wesley Publishing Co., 1983).

Hendrix, Gary G. (1977). "Human Engineering for Applied Natural Language Processing." *Proceedings of the 5th International Joint Conference on Artificial Intelligence.* vol. 1, August 22-25, 1977. pp 183-191.

Hendrix, Gary G., Earl D. Sacerdoti, Daniel Sagalowicz & Jonathan Slocum (1978). "Developing a Natural Language Interface to Complex Data." *ACM Transactions on Database Systems.* vol. 3, no. 2. June 1978. pp 105-147.

Horowitz, Ellis (1984). *Fundamentals of Programming Languages -- 2ed.* (Rockville, Maryland: Computer Science Press, 1984).

Ju, Charlie. (1984). *The Natural Language Front End Processor for Mistress Database, MS Thesis.* Department of Computer Science, Rochester Institute of Technology. 1984.

Kaplan, S. Jerrold. (1984). "Designing a Portable Natural Language Database Query System." *ACM Transactions on Database Systems.*" vol. 9, no. 1. March 1984.

Kelly, J. F. (1984). "An Iterative Design Methodology for User Friendly Natural Language Office Information Applications." *ACM Transactions on Office Information Systems.* vol

2, no. 1. January 1984. pp. 26-41.

Kolodner, Janet L. (1983). "Indexing & Retrieval Strategies for Natural Language Fact Retrieval." *ACM Transactions on Database Systems.* vol. 8, no. 3. September 1983.

Kuno & Ottinger (1962). "Multiple Path Syntactic Analyzer." in *Information Processing* (North Holland: 1962).

Leigh, William (1983). "Natural Language for Database Access." *Journal of Systems Management.* vol. 34, no. 8. November 1983. pp 22-24.

Li, Deyi. (1984). *A Prolog Database System.* (New York : John Wiley & Sons, 1984).

MITRE. (1964). *English Preprocessor Manual, Rep. SR-132.* Bedford, Massachusetts. 1964.

Petrick, S. R. (1965). *A Recognition Procedure For Transformational Grammars, Ph. D. Thesis.* Department of Modern Language, MIT. Cambridge, Massachusetts. 1965.

Rahmstorf, G. & M. Ferguson, eds. (1979). *Proceedings of a Workshop on Natural Language for Interaction with Data Bases.* (Laxenburg, Austria: IIASA, 1979).

Rhodnius Corp. *Mistress: Relation Database Management System User's Guide.* Rhodnius Incorporated. Toronto, Canada. 1983.

Riesbeck, Christopher K. (1982). "Realistic Language Comprehension." in Lehnert, Wendy G. & Martin H. Ringle. *Strategies for Natural Language Processing.* (Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1982).

Schneiderman, Ben. (1981). " A Note on Human Factors Issues of Natural Language Interaction with Database Systems." *Information Systems.* vol. 6, no. 2. 1981.

Shoval, Peretz (1981). "Expert/Consultation System for a Retrieval Data-Base with Semantic Network of Concepts." *ACM SIGIR Forum.* vol. 16, no. 1. Summer 1981. pp. 145-149.

Smith, Linda C. (1980). "Implications of Artificial Intelligence for End User Use of Online Systems." *Online Review.* vol. 4, no. 4. December 1980. pp 383-391.

Thompson, Frederick B. & B. H. Thompson (1975). "Practical Natural Language Processors : The REL System as Prototype." in M. Rubinoff & M. Yovitz, eds. *Advances in Computers, vol. 13.* (Academic Press, 1975). pp. 110-167.

Thorne, J., P. Bratley & H. Dewar (1968). "The Syntactic Analysis of English by Machine." *Machine Intelligence 3.* D. Mitchie, ed. (New York : American Elsevier, 1968).

Walker, D. E. (1981). "The Organization and Use of Information: Contributions of Information Science, Computational Linguistics and Artificial Intelligence." *Journal of American Society of Information Sciences.* vol. 32, no. 5. May 1981.

Waltz, David L. (1978). "An English Language Question Answering System for a Large Relational Database." *Communications of the ACM.* vol. 31. no. 7. July 1978. pp. 526-539.

Waltz, David L. (1982). "The State of the Art in Natural Language Understanding." in Lehnert, Wendy G. & Martin H. Ringle. *Strategies For Natural Language Processing.*

(Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1982).

Warren, David H.D. and Luis P. Pereira (1977). "PROLOG: The Language and Its Implementation Compared with Lisp." *Proceedings of the Symposium on Artificial Intelligence and Programming Languages.* vol. 12, no. 8. August 1977. pp. 109-115.

Weizenbaum, Joseph (1965). "ELIZA - A Computer Program for the Study of Natural Language between Man and Machine." *Communications of the ACM,* vol. 9, no. 1, January 1965.

Wilcox, Leonard E. Jr. (1983). *Parsing Natural Language, MS Thesis.* Department of Computer Science, Rochester Institute of Technology. 1983.

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

Winston, Patrick (1984). Artificial Intelligence, 2nd ed. (Reading, Mass: Addison-Wesley Publishing Co., 1984).

Winston, Patrick & Karen Prendergast, eds. (1984). *The AI Business: Commercial Uses of Artificial Intelligence.* (Cambridge, Mass.: M.I.T. Press, 1984).

Winston, Patrick H. & Berthold K. Paul Horn (1981). *LISP.* (Reading, Mass: Addison - Wesley Publishing Co., 1981).

Woods, William A. (1970). "Transition Network Grammars for Natural Language Analysis." *Communications of the ACM.* vol. 13, no. 10. (October 1970). pp. 591-606.

Woods, William A. (1977). "A Personal View of Natural Language Understanding." compiled by Waltz, David L. in "Natural Language Interfaces." *SIGART Newsletter.* no. 61. February 1977.