AN UPDATE EXPERT AND RESPONSE GENERATOR FOR A
TRANSPORTABLE NATURAL LANGUAGE INTERFACE TO
DATABASE MANAGEMENT SYSTEMS

by

Michael J. Bessasparis

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

MASTER OF SCIENCE

in

Computer Science and Applications

Approved:

Jane Terry Nutter, Chairman
Dr. Edward A. Fox
Dr. Rex Hartson

December, 1990
Blacksburg, VA
LD
5055
V955
1990
B 488
C.2
AN UPDATE EXPERT AND RESPONSE GENERATOR FOR A
TRANSPORTABLE NATURAL LANGUAGE INTERFACE TO
DATABASE MANAGEMENT SYSTEMS

by

Michael J. Bessasparis

Committee Chairman: Dr. J. Terry Nutter
Computer Science

(ABSTRACT)

Fully transportable natural language interfaces to database management systems (DBMS) have been under study for some years, but until now, all have suffered from a lack of response ability and lack of natural language update ability. Response generation is relatively easy to overcome, but the second problem, lack of update ability, is more serious. Adding update capacity involves primarily three tasks. First, the system must be able to recognize and process update requests. Processing an update typically involves both altering the knowledge base to reflect the new state of the database and performing dynamic extensions to the lexicon. Second, the intermediate language used to communicate with the database manager must be extended to cover update information. Third, the post-processor must be extended to transform commands into DBMS update requests.
The system described here uses a flexible and unified knowledge base to recognize and process update requests. Through information stored in the knowledge base, the system can recognize and resolve certain classes of ambiguity. The update request is then converted into an unambiguous intermediate query language. This language is easily translated to the target database management language using simple syntactic methods. The response generator uses the intermediate query language, the knowledge base, and the results returned by the target DBMS to form a response for all database accesses.
ACKNOWLEDGEMENT

I would like to thank Dr. J. Terry Nutter, my thesis advisor, for her extensive guidance, direction, and review of the thesis material. I’d also like to thank the other committee members - Dr. Ed Fox and Dr. Rex Hartson - for their review and comments. Last, but certainly not least, I’d like to thank my wife, Maria, for her patience and support.
# TABLE OF CONTENTS

## INTRODUCTION .......................................................... 1

## BACKGROUND ............................................................ 6
  Zero dimensions of transportability............................. 6
  One dimensional transportability................................. 8
  Two dimensional transportability................................ 10
  TIPS (Transportable Interface Prototype System).............. 17

## DESIGN ................................................................. 28
  Update Expert.......................................................... 29
    Update Parsing....................................................... 29
    Object References: Nouns and Modifiers........................ 30
    Verbs..................................................................... 33
    Determining Update Type............................................. 35
    Ambiguity Resolution................................................ 37
    Ambiguous Field Values............................................. 38
    Ambiguous Operations: Problems with Joins.................... 39
    Consistency Maintenance............................................ 43
  Extended IQL............................................................. 44
  Response Expert......................................................... 44

## RESULTS ................................................................. 46
  Update Expert.......................................................... 46
    Managing Ambiguity................................................... 48
    Consistency Maintenance............................................ 50
  Knowledge Base and Lexicon Maintenance........................ 51
  Intermediate Query Language........................................ 51
  IQL Post-Processor..................................................... 52
    Initialization........................................................ 54
    Additions.................................................................. 55
    Changes.................................................................. 55
    Deletions.................................................................. 57
  Response Generation..................................................... 59
  Novel Contributions of this Thesis................................. 61

## FUTURE WORK ............................................................ 63
  Extensions................................................................. 63
  Novel Implementations................................................ 64
    Enhanced Lexicon Management..................................... 64
    Error Handling........................................................ 65
    Enhanced Rule Enforcement........................................ 65
  Theoretical Work......................................................... 66
    Multi-Valued Fields.................................................. 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joins</td>
<td>67</td>
</tr>
<tr>
<td>Response Structure</td>
<td>69</td>
</tr>
<tr>
<td>User Modeling</td>
<td>69</td>
</tr>
<tr>
<td>CONCLUSIONS</td>
<td>72</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>74</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>78</td>
</tr>
<tr>
<td>APPENDIX A: Update Expert Grammar</td>
<td>78</td>
</tr>
<tr>
<td>APPENDIX B: Permanent Lexicon</td>
<td>80</td>
</tr>
<tr>
<td>APPENDIX C: Knowledge Base Structures</td>
<td>82</td>
</tr>
<tr>
<td>APPENDIX D: Update Expert ATN Diagrams</td>
<td>86</td>
</tr>
<tr>
<td>APPENDIX E: Update Expert Source Listing</td>
<td>92</td>
</tr>
<tr>
<td>APPENDIX F: Post-Processor Source Listing</td>
<td>135</td>
</tr>
<tr>
<td>APPENDIX G: Response Expert Source Listing</td>
<td>155</td>
</tr>
<tr>
<td>APPENDIX H: Sample System Output</td>
<td>169</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Database management systems (DBMSs) provide efficient storage and retrieval for large bodies of information used by people. But to interact with most databases, users must learn a formal DBMS language. People who are not expert computer users find such languages hard to learn and difficult to manipulate. Natural language interfaces (NLIs) give nonexpert users the ability to work with this efficient system of information storage.

NLIs can be classified by how transportable they are. Transportability has two dimensions: the ability to port the NLI across data domains easily and across DBMSs. Early NLIs (SHRDLU [Winograd 1972]) were tied to particular data domains and to particular DBMSs, and therefore have zero dimensions of transportability. The next generation (TED [Hendrix and Lewis 1981]) began to provide portability across domains, and recently (TEAM [Grosz, Appelt, Martin and Pereira 1987]) full two dimensional transportability has been achieved. Although these NLIs can be ported across data domains and DBMSs relatively easily, they have two obvious drawbacks.

First is the lack of natural language response ability. Current fully transportable interfaces return raw data to the user. While this might be preferred in some cases it
does nothing to make the uninitiated feel comfortable using the machine. This shortcoming is relatively easy to overcome but the second problem, lack of update ability, is more serious.

Current NLIs let users query a database but do not let them update it. Clearly many databases are dynamic, and this ability cannot be dismissed as unimportant or reserved only for the database experts. Two problems are evident when dealing with updates: flexibility of the knowledge base and ambiguity in the update sentence.

A transportable NLI that is capable of processing updates to the database must be flexible. It must have the ability to model the current state of a constantly changing database. This requires a knowledge base and lexicon that are easily accessible and flexible. In current systems (TEAM [Grosz, Appelt, Martin and Periera 1987]), these components are static or very difficult to modify. Furthermore, the parsing mechanism must be flexible enough to deal with ambiguity.

Ambiguity is encountered in at least two forms: underspecified references and ambiguous operations introduced by the use of joins. Underspecified references may be encountered when parsing either new words or words that can have multiple meanings. The parser must make
assumptions about the meaning of new words by examining the context. After these new words are defined they must be entered into the knowledge base and lexicon. The user can then use these words as if they had been defined when the system was originally installed.

Many times a database contains specific values that may refer to more than one field. When a user doesn’t know about the multiple senses the interface must recognize and resolve this ambiguity. The user’s limited knowledge of the whole database also introduces operational ambiguity.

Because the user of a NLI is separated from the database, he is probably not familiar, in detail, with its structure. This structural unfamiliarity means that the user’s view of how the data is organized is not the same as its actual organization. Because of this, the user may specify updates that, in his view, are simple and unambiguous, but that can be accomplished by altering the actual database in various ways. The NLI must be able to map the language of the user, which may be operationally ambiguous, to the actual structure of the database.

This thesis examines these issues. An update expert and response generator are then designed. The design is validated by extending an existing transportable interface. The chosen architecture is well suited for updates because
of its unified and flexible knowledge base and lack of reliance on a syntactic parser.

Updating of the database is performed by recognizing an assertion made by the user, parsing the assertion into an intermediate query language, and finally translating the intermediate language to a database update in the target DBMS language. The expanded interface recognizes semantic ambiguity and disambiguates with a minimum of user assistance. During the parse the knowledge base and lexicon are updated to reflect new information. After the database operations have been performed, control is passed to the response generator.

The response generator uses the intermediate query language and knowledge base to generate an appropriate response. The generated response has a very simple structure and is independent of the original sentence. This simple unambiguous response allows the user to verify that what the system did was what the user intended. The response can also be understood by the interface. Because the system can understand the response the user is guided in forming other sentences.

Chapter two of this thesis discusses some past natural language interfaces concentrating on their transportability, update ability, and response structure. Chapter three
introduces the proposed structure of the update parser and response generator. Chapter four presents one implementation of this solution and chapter five proposes further enhancements.
2. BACKGROUND

Building NLIs for computer systems is not a new idea. As long as computers have been used to store information, people have been striving to make this information easily accessible and manageable. As the relational database became more popular, efforts were concentrated in building NLIs to these systems. As these NLIs became more sophisticated, the cost in time and effort, of developing a new interface for each new database or for each new DBMS rose. One front-end which could easily be transported across many different databases and data managers was most desirable. Such an interface would incur transportation costs but these are typically much less than the costs associated with building a new interface for each new database or new data manager. The gradual evolution from non-transportable to fully transportable is apparent in past systems such as SHRDLU, TED, ASK, and TEAM.

2.1. Zero dimensions of transportability

SHRDLU was developed for use in the "blocks world." This interface is designed to be used on one set of data (blocks) with one data manager (itself). It is impossible to use SHRDLU to interact with any other database or data manager because the interface is both the database and the DBMS. A
system like this that is not portable over databases or DBMSs can be said to have zero dimensions of transportability.

The SHRDLU system was designed to model a robot arm moving objects around on a table. The system uses a model of the "blocks world" to reason about queries. For example one might ask it "Why did you pick up the green pyramid?" to which it could respond "To clean off the red cube." Updates are a large part of the "blocks world" system. These updates are in the form of imperatives such as "Pick up a big red block," and declaratives such as "The blue pyramid is mine."

The system handles these updates well in part because the syntax, the semantics, and the pragmatics of the sentence are all considered concurrently as the sentence is being parsed. This integrated processing of the sentence enables the system to apply semantic and pragmatic tests as the sentence is parsed as well as resolve certain classes of anaphora. Such processing is straightforward because the interface has direct access to general knowledge about the domain and to specific knowledge about the current state of the world.

As mentioned above, SHRDLU also generates responses. Both of these operations, accepting updates and generating
responses, are straightforward when the database, the data manager, and the interface are all one. These operations become more complex when greater transportability is introduced. This complexity is due to the separation of the interface from the database. It is the lack of this separation that ties SHRDLU to one database and to one DBMS.

Other non-transportable systems include LADDER [Hendrix, Sacerdoti, Sagalowicz, and Slocum 1978], PLANES [Waltz 1978], and JETS [Finin, Goodman, and Tennant 1979]. LADDER is one of the earliest semantic grammar based systems. A semantic grammar is concerned with the meaning of words such as ship and dock rather than syntax, such as noun and verb. PLANES and its successor, JETS, are two systems that consider methods for dealing with incomplete sentences from the user.

2.2. One dimensional transportability

Interfaces which are transportable across data domains but not across managers have been around for about a decade. In 1981, researchers at SRI reported on a prototype for transportability across domains, the Transportable English Datamanager (TED). Other such systems include PRE [Epstein 1985], Datalog [Hafner and Godden 1985], and IRUS [Bates and Bobrow 1983]. Domain transportable systems typically
guarantee portability by one of two strategies. The first
strategy, which IRUS, Datalog, and PRE use, isolates the
portion of the semantics (and in Datalog's case of the
syntax) which is domain-dependent, and is reprogrammed as
needed. This requires changing the essential working code
of the system, but usually in a limited and controlled
fashion. This approach presupposes that porting across
domains is the exception rather than the rule.

The second strategy, used in TED, implements an interface
expert which interviews the system administrator to produce
the necessary domain-dependent information. The interface
expert questions a human data manager about the structure of
the database and about the data domain. The answers to
these questions are used to build a knowledge base that is
separate from the actual TED interface.

After this background knowledge is processed, the user may
then build the database. This building must take place in a
language similar to conventional database systems. When the
database has been loaded, the user can then query the system
in natural language. The parser accepts these queries and
builds a formal query, using the background knowledge from
the interview and accessing the actual database directly.
This formal query is then used by a built-in data manager to
retrieve the specified information and display it for the
user to examine.
Although TED fulfills its design goal, transportability across data domains, it has three fundamental shortcomings. First, it is tied to one data manager. Second, the user must use a formal database management language to update the underlying database. Finally, the system merely displays the information from the database for the user - it doesn’t actually respond, in natural language, to the user.

2.3. Two dimensional transportability

ASK provides an interesting intermediate case. ASK is a data manager for a DBMS which lets the user query a foreign database. The user explicitly requests foreign access. ASK calls an interview expert which queries the user on all information needed to access that system, from the foreign system’s query language and database structure to physical details such as hardware type, baud rates, etc. Using this information, it translates the user’s query into the foreign language, configures itself as a terminal, queries the foreign system, and dumps the result as unprocessed text.

This is ingenious, but it has an obvious disadvantage. To query the foreign system successfully, the user must know its hardware and software features in detail and be able to formulate its queries. Such a user could get the same
information faster and more easily by going to another terminal, logging on to the other system, and asking for it.

ASK's limitations relative to foreign data managers results from its deep integration with its own system: like TED it is the data manager. To achieve portability across DBMSs, interfaces must be decoupled as far as possible from the data manager. A system exhibiting this kind of portability is TEAM. The architecture of the TEAM system is illustrated in figure 1.

TEAM isolates a layered knowledge base containing a conceptual schema, a database schema, and a lexicon from the parsing mechanism. The conceptual schema contains information about the objects and relations in the domain of the database. It also contains constraints on the arguments passed to it as pragmatic information. This information is stored in a sort hierarchy composed of levels of detail about objects. For example, the sort hierarchy represents knowledge that height is a linear measure, that linear measure is a type of measure, that measure is a scalar abstract object, and that an abstract object is a type of thing.
Figure 1: Architecture of TEAM (Adapted from [Safigan, 1987])
The database schema contains information about the exact structure of the underlying database. This information is needed to translate the logical form produced by the parser to a SODA [Moore 1979] query. The database schema contains a list of key values for each relation, definitions of predicates and the fields and relations to which they refer, a list of identifying fields and the sorts they belong to, as well as definitions of the sorts in terms of the database fields and relations.

The lexicon contains the words used to interact with the system. It is made up of two classes of words: closed and open. Closed class words are very general and can be used in almost any domain. This includes words such as pronouns, conjunctions, and determiners. These words generally serve very specific grammatical roles. Open class words are very dependent on the particular domain. These words include names for fields and particular values for these fields.

TEAM also takes a layered approach to query translation. Given a natural language query, TEAM translates the query into a parse tree, converts the parse tree into a logical form (an extension of first order predicate calculus), then uses a schema translator to transform the logical form into an expression in an intermediate query language called SODA. A post-processor then translates the SODA query into a query
in the language of the actual DBMS. Hence when the interface is transported to another DBMS, only the code that translates the SODA query must be rewritten.

TEAM distinguishes three different field types: feature, arithmetic, and symbolic. A feature field contains boolean values that indicate whether or not a given feature is present. Arithmetic fields are fields that hold numbers and as such can have special operations performed on them, i.e. addition, averaging, etc. A symbolic field holds alphanumeric values. These values describe attributes and typically correspond to adjectives or nouns.

TEAM supports the use of virtual relations. Virtual relations allow the system to make connections between actual relations. For example, one relation may hold the names of mountains and the countries where they are located. Another relation may hold the continents where each country is located. By using both of these relations one can find out on which continent a particular mountain is located.

TEAM has the capability to deal with a wide range of user defined verbs. These verbs are in addition to the verbs to be and to have. It is the use of these verbs that gives TEAM a more complete range of acceptable input queries. This also allows the user to interact with the interface in a more natural way. For example, the user may query the
system with "What countries have an area greater than 3 million square miles?" Clearly this is sufficient, but a more natural query might be "What countries cover more than 3 million square miles?" To process this query TEAM requires knowledge about verbs.

TEAM acquires knowledge about verbs during the interview process by observing how the person being interviewed uses the verb and by accepting criticism of its use of the verb in sample sentences. For example, a user may want to define the verb *cover*. He may use it in a sentence like "A country covers an area." TEAM then asks the user if "An area covers" is the same as "Something covers an area." The answer is no. TEAM now knows that *cover* is transitive and must have two arguments. The system next wants to know if "A country covers" is the same as "A country covers something." This is no for the same reason. Finally, "An area is covered" is the same as "Something covers an area." This indicates that the verb may be used in the passive.

Neither ASK nor TEAM has any natural language response facility, and neither one manages updates. This is typical of current natural language front-ends. The lack of response facility is relatively minor and easy to remedy. The inability to handle updates is more crucial. Essentially, most front-ends achieve their effects by maintaining knowledge bases that reflect vital facts about
the database, including all the vocabulary that is needed to talk about it. These knowledge bases are typically static. Managing updates would require keeping the knowledge base current with regard to the present system state.

Further complicating the issue of updates in TEAM is the complexity of the knowledge base. Even if one could devise a way to update such a layered knowledge base quickly and easily, the syntactic parsing of sentences remains a problem. TEAM is limited by the need to generate a parse tree as the first step in the acceptance process. Complete knowledge of the syntax used in the sentence is needed. Because of this, the parse fails when it is presented with a word not in the lexicon.

In real world situations, a front-end which cannot survive alterations to the database is little more useful than one that cannot be used for more than one application. Certainly it is not difficult to port these interfaces across versions of the database; but two points arise. First, such a process presupposes that all persons who submit updates to the database are necessarily experts in the use of DBMS languages; this assumption is already unreasonable, and will become more so rapidly. Second, there is a cost associated with porting, and it must be borne every time a change is made. This adds up quickly.
If updates can be performed through the front-end, no reprogramming is needed, and the cost goes away.

2.4. TIPS (Transportable Interface Prototype System)

TIPS [Nutter, Safigan and Diaz 1989] is a prototype architecture for a fully transportable natural language interface to relational database management systems. From its outset, TIPS was designed to support full transportability across both domains and managers, and with the intent of supporting updates and natural language responses. TIPS includes three major modules: a knowledge base, an interface expert, and a semantic parser. The parser translates the user's input directly into the TIPS Intermediate Query Language (IQL). Small post-processors manage final translation from IQL to the target DBMS languages. Figure 2 gives the overall TIPS architecture. The shaded modules indicate parts of the system that are designed and implemented in this thesis.

The knowledge base is a major key to the success of the TIPS design. This is where all the system's information about the database structure and data domain is stored. This knowledge base is constantly accessed by the different modules in the system. Hence, the information must be stored so retrieval from and updating to the knowledge base
Figure 2: Architecture of TIPS
is both easy and fast. To let the interface parse an input correctly and generate IQL, the knowledge base must contain information about the data domain and the structure of the database.

Knowledge of the data domain enables sentence parsing. This knowledge includes the identity of key values, corresponding adjectives, and the fields referred to by user defined verbs. It is this knowledge that gives a sentence meaning. For example, given that the system knows that patients see their doctors it can parse the sentence "Who does Mary see?" and understand that the user wants the value that is in the doctor field of the record with the key field Mary.

Knowledge of the structure of the database is of primary importance when the interface is building the IQL. Having this knowledge at hand enables the interface to transform the view that the user has of the database to the actual database structure. For example, the user might be interested in patient names and their corresponding diets. Suppose that the database is organized so that patients have a diagnosis, and that in a separate relation each diagnosis is associated with a particular diet. See figure 3. The user asks for Mary Smith's diet. The interface knows about the separate relations and formulates a query so that Mary's diagnosis is found, and then the diet corresponding to that
<table>
<thead>
<tr>
<th>Patients</th>
<th>Name</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mary Smith</td>
<td>diabetes</td>
</tr>
<tr>
<td></td>
<td>Tom Jones</td>
<td>ulcer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Care</th>
<th>Diagnosis</th>
<th>Diet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>diabetes</td>
<td>ADA</td>
</tr>
<tr>
<td></td>
<td>ulcer</td>
<td>bland</td>
</tr>
</tbody>
</table>

Figure 3: Example Database
Figure 4: Example SNePS Structure
diagnosis is returned to the user. The user need not be concerned with the organization of the data.

In TIPS domain and structural information is obtained by the interview expert and stored in a lexicon and semantic network using SNeps [Shapiro 1979; Shapiro and Rapaport 1986]. The information is stored in a straightforward manner. For example, the fact that age is a field in the patients database is represented by the SNeps structure given in figure 4. This figure also represents the facts that age is a numeric field and that old is a maximizing adjective that can be used to describe age, i.e., "The oldest person." The use of a semantic network allows for rapid recovery of information from the network, easy modification of the knowledge base, and first order logic inference capability.

It is essential that the knowledge base be flexible if it is to continue to represent the current state of the database. A semantic network provides this flexibility. It is also essential that all of the knowledge in the system be unified. Unification enables the different modules of the interface to use the same information for different purposes. As an example, the query parser uses the units of numeric fields to determine which field to access while the response expert uses the same information to generate a more
meaningful response. Again, the use of a semantic network enables the interface to store different kinds of information in one place.

The interface expert itself currently contains one module: the interview expert. The interview expert is designed to interview a human data manager about the structure of the database. This interview takes place only once for each database. During the interview, the system learns about the names of databases, names of fields, units of measure, maximizing and minimizing adjectives, modifiers, and, database joins. As the human data manager answers questions, the system builds the knowledge base. When the interview is complete the system is ready to accept natural language queries. Figure 5 shows a portion of an interview session with the corresponding structures that are built in the knowledge base.

TIPS, like TEAM, supports three types of database fields: alphanumeric, numeric, and boolean. It is the task of the query parser to map the natural language query into these types of fields. The query parser is an augmented transition network (ATN) based semantic grammar. The parser accepts natural language queries, parses them using the information about the database structure in the knowledge base, and produces an IQL query. The query parser produces no new network structures. In other words, as the ATN
(Input from the user is underlined)

Enter new database filename or end: personnel-file
Done! Enter synonym for personnel-file or end: employee
Done! Enter synonym for personnel-file or end: end
Enter field name or end: employee-name
Enter synonym for employee-name or end: end
Is this a key field? y
Is field (A)lpha, (N)umeric, or (B)olean? a

Figure 5: Example Interview and Corresponding Network
accepts the query, it is discarded. Therefore, after the entire query has been parsed, the system has only the IQL representation of the original query. The IQL query passes to a post-processor, which translates it into a series of commands in the language of the target DBMS.

An intermediate query language used to support transportability across DBMSs must meet three requirements:

1. It must be unambiguous.
2. Its command specifications must correspond to and unambiguously determine the organization of the data in the target database.
3. It must not require operations that cannot be performed by the target DBMS. Since the target DBMS is not known in advance it should require only basic operations that are widely available on commercial DBMSs.

IQL is an extremely simple intermediate query language. It consists of non-empty lists of up to four registers (variables): a retrieval register, a qualification register, an operation register, and a join register. In short, the retrieval register contains the specification of which fields to retrieve from which files in a <filename> . <fieldname> form. The qualification register serves to restrict the operation being performed. The operation register describes the action to be taken. These actions include list, yes/no, count, etc. The join register
supplies the linking information. This linking information is in the form (from-file . from-field to-file . to-field). The information in these registers can be translated to formal queries in almost any standard commercial DBMS language using only simple syntactic methods. An example of an interaction with the system taken from Safigan [Safigan 1987] follows (user input is underlined):

: *give me the number of employees who work in the data processing department with not more than 150 dollars of whole life costs*

Contents of registers:
retrieval: (personnel-file . employee-name)
qualification: (insurance-rates . insurance-costs <= 150 and personnel-file . employee-department = data processing)
operation: count
join: (insurance-rates . insured-age personnel-file . employee-age)

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

In this example one can see the relationship between the original query and IQL. Furthermore, the nature of the translation from IQL to a DBMS language, in this case Model 204 [Model 1983], is purely syntactic and, thus, very fast. The post-processor that translates from IQL to a target DBMS language is the only part of the system that must be rewritten when it is transported from one DBMS to another.
The cost of transporting the interface to a new DBMS is the cost of developing this post-processor for the new DBMS. Because of this IQL has been designed for easy transformation into actual DBMS commands. This ease of translation has been demonstrated by the implementation of a post-processor that translates IQL into commands that may be executed in the McMax [McMax Reference Manual 1987] database environment [Nutter, Safigan and Diaz 1989].
3. DESIGN

As mentioned above, TIPS was designed to allow for an update expert and a response generator. These were not included in the original version. The current work presents a design and implementation of an update expert and a response generator. To further demonstrate the flexibility of IQL, a third translator from IQL to the dBase III [Hergert 1985] database language is also implemented here.

When the system identifies an update request from the user it passes control to the update expert. The update expert fills IQL registers while parsing the update request. The update expert also must make the required changes and additions to the knowledge base, as well as update the dynamic lexicon. Like the query parser, the update expert must produce IQL that will be passed to the post-processor and then to the response generator.

The response generator forms a response from IQL. This approach is taken because it is fast and easy. Furthermore, the response generated will be unambiguous: it will reflect the actions taken by the underlying DBMS without the usual ambiguity inherent in natural language.

The update expert and response generator designed here are prototypes. They are intended to demonstrate the viability
of the underlying ideas. They do not attempt to encompass all natural language interactions. If wider language coverage is required, the modules can be extended. Nevertheless, the subset of updates accepted and the responses generated are reasonably expressive. The update expert grammar is given as appendix A.

3.1. Update Expert

Accepting an update involves three phases. The first phase is the actual parsing of the update request. While in this phase, the semantics of the update request are distilled into IQL. During the second phase, any ambiguity that may have arisen during the parse is resolved. The final phase of the update expert, consistency maintenance, is required to insure that the update will not violate semantic constraints placed on the database by the database administrator.

3.1.1. Update Parsing

Parsing an update request is similar to parsing a query. The parser accesses the knowledge base and lexicon to translate the meaning of the natural language input into IQL. Unlike query parsing, update parsing requires the parser to make assumptions about the meaning of words that
are not in its lexicon and about concepts that are not in the knowledge base. Additionally, the parser must modify the lexicon and knowledge base to insure consistency with the actual database. Clearly, it would be very time consuming to implement a parser for all updates that can be expressed in natural language. Therefore, this work concentrates on a particular class of update sentences.

3.1.1.1. Object References: Nouns and Modifiers

Many sentences people use to refer to objects in a relational database have a simple two-part structure. The initial noun phrase qualifies the objects of the update, while the following verb phrase provides the new or changed values. For example, in the sentence "Peas are green," the initial noun phrase describes objects, peas, and the verb phrase, are green, indicates a new or changed value. The sentence structure "<qualifications> MAIN VERB <updates>" serves to describe a broad range of updates to a database. Both <qualifications> and <updates> may be compound. For example, "Graduate students who have earned a QCA of at least 3.5 are honor students." In this sentence, the students are qualified by the phrases graduate and QCA of at least 3.5. This set of students is assigned the value honor to fill some attribute.
With some restrictions, sentences of the above form can be mapped into DBMS commands. In defining this mapping, we look at the three major components of these sentences: subjects, verbs, and modifiers.

Subjects of sentences that refer to objects stored in a database map very well into the names of relations in those databases. Some examples of this behavior are "Red cars . . . .," "Ships that have missiles . . . .," and "Pregnant women . . . ." For each of these sentences we can imagine database relations that are named cars, ships, and women. So, we see that we can map the head noun of a noun phrase into the name of a database relation. Next, we consider modifiers.

Modifiers, both adjectives and nouns, map very well into field values of relational databases if one constraint is applied: all modifiers must refer to the head noun. In a normalized relation, one in which all attributes serve to describe a particular property of the key, this is not a problem. Given this constraint, phrases like "red chairs" and "stroke victims" make sense. The above adjectives and noun-modifiers refer to alphanumeric values. Adjectives also may refer to numeric fields.

Numeric fields may be modified by two special types of adjectives: maximizing and minimizing adjectives. [Safigan
1987]. As the names imply, they are used to refer to the greatest and least values of a numeric attribute. For example, "The oldest patient . . . ," or "The lightest ship . . . ."

If the adjectives old and light have been defined to refer to age and weight then the above phrases make sense.

In this system, the use of maximizing and minimizing adjectives is restricted to the qualification phrase of the sentence. Although one can devise somewhat reasonable actions to be taken if these adjectives are used in the update phrase, as in "Mary is the oldest patient," this is unwise for at least two reasons.

Let's say that the above sentence means to find the oldest patient and make her name Mary. This results in a change being made to the key field of the relation. As discussed later, this is dangerous.

A second problem is apparent when Mary's record must be added to the relation. The semantics may be to find the age of the oldest patient and make Mary's age some value greater than that age. This may sound reasonable at first glance, but there now exists a particular value for Mary's age that is incorrect. Furthermore, there is no way to determine that this is an incorrect value. Unless the interface can deal with uncertainties, which this prototype cannot, the
values in the database are misleading. It is better to restrict the sentence semantics than to mislead the user.

3.1.1.2. Verbs

When studying objects that may be stored in a relational database, the only verb that is absolutely required is to have. All relational databases have tuples that are made up of attributes. These attributes have specific values. It follows that one may update anything stored in a relational database by simply identifying a set of target tuples, followed by the verb have, the update value, and the attribute name. As a variation, the sentence structure may consist of an identified tuple set followed by the verb have, an attribute name, the preposition of, and the update value. For instance, using the above rules we may assert that "Battleships have Exocet missiles," or "Mary’s car has a color of red," or "Susan has a pregnant of true."

This last sentence illustrates a problem with including only this one verb. Speaking of someone having a pregnant of true may make sense when referring to an entity stored in a relational database, but it is silly when referring to actual people. So, while this single verb would suffice, it would make for unnatural and cryptic interactions. The
ability to handle more verbs is required of a "natural" language interface.

The next common verb to consider is to be. The verb to be maps values into fields in two ways. First, Boolean values can be easily mapped into Boolean fields using to be. This solves the above problem of dealing with pregnant. If pregnant is a Boolean field, the sentence "Mary is pregnant" now has meaning to both the database system and a person not schooled in relational theory. Specifically, the sentence asserts that Mary’s record in the database has true in the pregnant field. Analogously, to map the value of false into the pregnant field, one can say "Mary is not pregnant."

The second way that to be is used involves more general field values. In this role the verb to be maps predicate adjectives, predicate nouns, and numbers into appropriate fields. For example, in the sentence "The chair is red," the value red can be mapped into a field, say color, describing chairs. To avoid potential ambiguities another form of this sentence might be "The chair’s color is red." Similar constructs are used when the predicate is a noun. For example, to say that a particular patient has been diagnosed as suffering from acute-mi one could say "Fred’s diagnosis is acute-mi." Finally, to map numbers into numeric fields one might say "Tom is 22 years old." Note in this last example that the units of measure become important
in determining what field the value 22 refers to. The maximizing adjective old also may be used to determine which field is being referred to.

The above two verbs provide for an impressive amount of description, but people generally use many more verbs. Some of these verbs are used to shorten sentences when the proper background knowledge is present. For example, if one knows that employees earn salaries then the sentence "Susan earns 20,000 dollars" means the same thing as "Susan has a salary of 20,000 dollars." At least both mean the same thing in terms of a mapping into a relational database. Specifically, both mean to find Susan's record in the employees relation and make the value of the attribute salary 20,000.

3.1.1.3. Determining Update Type

Now that we can map natural language constructions into database structures, we must have a way to determine the action type, i.e., add, change or delete. A new record is added to the database by defining a new key value. It is the fact that this key value is not already in the database that distinguishes an add from a change. After the key value is defined the remaining attributes may be described as above. Some examples of additions are:
Mary is a female patient and she sees Dr. White. John has a diagnosis of acute-mi. Fred was admitted with chest-pain. Susan is a white female patient who was admitted with chest-pain.

Two records in a relational database may not have the same key value. Therefore, if a reference is made to a key value that already exists one must be making a change in the database. A change also may be made by referring to specific attributes of records. This reference is restricted by the fact that these attributes must be defined, i.e., one may not qualify a set of records using unknown attribute values.

After the target records have been defined, the new or changed values may be introduced, with one restriction: it is not possible to change the value of a key field. If it were possible to change the key field, then the database could quickly become inconsistent, i.e., have non-unique key values. In this system, the only valid way to change a key value is to delete the record and then to add it with the new key value. Some examples of changes that follows these guidelines are:

Female patients see Dr. White.
Fred has a low-cal diet.
Patients with an age greater than 80 years are geriatric.
Unlike an addition or a change, a deletion command need only be qualified. The target records may be qualified in two ways. First, by reference to a particular key value. Secondly, by reference to a range of records using the field values to define the records to be deleted. For example:

Delete married patients.
Delete Mary.
Delete foreign cars that have a cost greater than 40,000 dollars.

3.1.2. Ambiguity Resolution

Natural language contains many ambiguous structures. Ambiguity occurs in natural language updates when those updates contain underspecified references. The DBMSs that are currently available are unable to execute a sequence of commands correctly unless the data manager can determine the exact correspondence between relations, values, and fields. In general, DBMS languages specify this correspondence explicitly by including the relation name, field name, and the field value in a command. Conversely, people generally imply these references and assume that the listener resolves the reference either through examining the context, referring to background knowledge, or by asking the speaker. Therefore, the natural language interface must disambiguate any underspecified references present in the update before a proper IQL command is built.
This prototype update expert does not attempt to deal with all the ambiguities that arise in natural language updates. However, it does consider two of the most common: ambiguous field values and ambiguous operations.

3.1.2.1. Ambiguous Field Values

Field values may be ambiguous for two reasons. Either the values refer to two or more fields in the database or they have never been defined. In either case the interface can use background knowledge and context to disambiguate. If these fail then the interface must consult with the user.

For example, say that in a particular database one may describe both cars and flowers in separate relations named cars and flowers. Further, say that both have an attribute color that may take a value of red. This information is stored as background knowledge in the knowledge base and is accessible to the parser. By using background knowledge and phrasal context, the interface can determine that it should access the color field in the cars relation if a user refers to "red cars" and the color field in the flowers relation if the user refers to "red flowers."

When an undefined value is used in a sentence, the interface may eventually need to consult with the user to determine what field the value refers to. This consultation is kept
to a minimum by using sentential context to narrow the range of possible fields. For example, assume that the interface has never encountered the value Ford. If the user asserts that "The pick-up truck is a Ford," the interface must ask the user what Ford refers to. Immediately, the range of possible fields has been narrowed by recognizing the inclusion of pick-up in the update, i.e., Ford probably does not refer to class-of-truck.

By including the ability to resolve limited ambiguity, the interface has reduced the intellectual demands placed on the user in connection with formulating a well specified query. The user can concentrate less on the exact content of the database and more on his particular task. This is one goal of natural language interfaces. Another intellectual burden placed on the user by many systems is knowledge of the structure of the database.

3.1.2.2. Ambiguous Operations: Problems with Joins

A database may consist of many different relations. These relations may be joined in meaningful ways to form virtual relations. This is commonly done with many-to-one relations to reduce the amount of storage needed. While storage is reduced, the complexity of updating these joined relations is increased.
When updating a database that consists of multiple relations, a user may easily specify updates that are operationally ambiguous. This ambiguity comes from the user not including the join fields in his update specification. This may occur because the user does not know the specific structure of the database he is using, but also may occur simply because he forgot. An example clearly illustrates the problem.

Suppose that two relations are defined as in figure 6a. Furthermore, suppose that these relations are joined by the department field. It is clear that Smith is managed by Fisher and that the manager of Jones is Meyers. Although there is no problem parsing a query, ambiguity may be introduced when updating these relations. For example, what actions should be taken if the user asserts that "Smith's manager is Meyers"? There are two choices.

One choice is to access the employees relation, determine that Smith's department is sales, access the managers relation, find the sales department, and then change the manager of this department to be Meyers. This has made Smith's manager Meyers, but has had the side effect of also making Clark's manager Meyers.

The second sequence of operations is to retrieve the department managed by Meyers from the managers relation,
<table>
<thead>
<tr>
<th>Employees</th>
<th>Name</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Sales</td>
<td></td>
</tr>
<tr>
<td>Clark</td>
<td>Sales</td>
<td></td>
</tr>
<tr>
<td>Jones</td>
<td>Advertising</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Managers</th>
<th>Department</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
<td>Fisher</td>
<td></td>
</tr>
<tr>
<td>Advertising</td>
<td>Meyers</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6a: Example Database

<table>
<thead>
<tr>
<th>Employees</th>
<th>Name</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Fisher</td>
<td></td>
</tr>
<tr>
<td>Clark</td>
<td>Fisher</td>
<td></td>
</tr>
<tr>
<td>Jones</td>
<td>Meyers</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6b: Example View
find Smith's record in the employees relation, and change the department field to be that managed by Meyers, in particular, advertising. This sequence of operations has no side effects.

One solution to the ambiguity problem is to recognize that both operations yield the result specified by the user and to ask the user to choose between them. For example, the user may be presented with the following:

By "Smith’s manager is Meyers" do you mean:

a) make Meyers the manager of the sales department  
b) move Smith to the advertising department

This approach assumes that the user is aware of the structure of the database. This presents a problem if the user views the database as in figure 6b. If we ask a user with this view to choose either update a or b he will surely be confused by the reference to the department field. We would like an approach that assumes only that the user views the database as expressed in his update.

It has been argued by Keller [Keller 1982] and by Davidson and Kaplan [Davidson and Kaplan 1983] that, given such a view of the database, the update that causes the least number of side effects should be made. In this case, the second sequence of actions should be taken.
The ability to disambiguate actions frees the user from specifying, or even knowing, the exact structure of the database or the exact procedure needed to carry out the update. If a user is unaware of the exact values in the database and the exact structure of the database also may be unaware of any semantic constraints on the values in the database. The final phase of update parsing, consistency maintenance, is designed to alert the user of violations of these semantic constraints.

3.1.3. Consistency Maintenance

Traditional relational database management systems include very little consistency checking facilities. Type-checking is common. While type-checking is an important guard against incorrect information being entered into a database it is not enough. There must be some way to insure that values of the correct type make sense. Range checking is a good way to do this.

Range checking requires that someone, a database manager, say, determine the correct range of values of certain fields. For example, the data manager may decide that the values acceptable in the age field must be between 0 and 150. Given this information, the interface monitors all updates to the database to insure no values outside of this
range are given. If such a value is given the interface alerts the user and asks for another value. Occasionally, a rule may be required to be broken. So, after the user is alerted to the illegal value it is assumed that he will enter the correct value. This reentered value is not checked against the rules, i.e., the built in rules may be overridden.

3.2. Extended IQL

The extended IQL must support all three types of updates. In order to support adds and changes properly, a new register is needed. This new register holds the new or changed values. In addition to this new register, the semantics of IQL have been extended to include support for adds, changes, and deletes. In parallel with extending the semantics of IQL, the post-processor also is extended to translate updates from IQL to a target DBMS language.

3.3. Response Expert

The final part of the system is the response expert. The response expert must generate a response for the user from IQL. Generating the response from IQL, as opposed to generating from semantic network, is preferred for at least two reasons: simplicity and precision.
Generating the response from IQL is simple. The meaning of the user’s input has been distilled into IQL, so it is there anyway, and may as well be used for more than one task. Secondly, IQL describes the precise actions that are performed on the database. This allows the user to see exactly what has happened to the database. Such a response will tend to be verbose because of the precise nature of IQL. This wordiness is acceptable to keep the user informed of the state of the database. For example, the interface may generate a response such as "I understand that Mary has a race of white." A less verbose response is "I understand that Mary is white." While the second sentence is more concise, it does not show the user exactly what field white refers to. Another advantage related to the unambiguous nature of IQL is that the system produces responses that are unambiguous. These responses can be understood by the interface. This provides cues to the user so that ambiguity may be removed from future sentences, thus making the interaction process faster and easier.
4. RESULTS

This project extended the previous version of TIPS by implementing the update expert, the extended IQL, a third post-processor, and the response generator. These extensions are implemented in SNePS version 2.1. In order to minimize the amount of support work needed, the system is heavily dependent on the facilities provided by SNePS. The update expert and response generator are implemented in the augmented transition network provided in SNePS. This simplifies accessing the knowledge base, which is implemented in the semantic network inherent in SNePS. The SNePS lexicon is also used by the interface. The remaining part of the interface, the post-processor, is implemented in the Allegro Common Lisp version 3.1 that underlies SNePS. The entire system is running on a NeXT computer.

4.1. Update Expert

The update expert is implemented using the grammar found in appendix A. A complete listing of the arcs and nodes of the update expert ATN can be found in appendix E.

When the interface encounters a sentence whose first word is unknown, it assumes that this sentence represents a request for an addition. In such a request the unknown first word must be a new key value. This new key value is added to the
knowledge base and the lexicon. The grammar then provides for a verb to follow the key value. The verb may be a form of BE, HAVE, or a user defined verb (provided through the interview expert session). In each case the object of the verb or preposition indicates how the fields of the new record are to be filled and how the knowledge base and lexicon must be updated. The strategies that are used to determine the appropriate fields for each of the values given are discussed below.

An update request in which the subject of the sentence is known is assumed to be a request for a change. This subject can take the form of a known key value, known field value, or a pronoun. In the first two cases the qualification register is filled by parsing the given subject. In the case of a pronoun the previous sentence’s qualification is used. For simplicity, no attempt is made to insure the previous qualification agrees in number or gender. Relative clauses or prepositional phrases can be used to further restrict the subject. As in the case of a database add, a verb must follow the subject, and the object or preposition of the verb determines the new or changed field values.

Requests to perform deletions begin with the word delete. The subject of the sentence must follow delete and the qualification register is filled by continuing the parse. The subject must be well defined as above.
4.1.1. Managing Ambiguity

Through information stored in the knowledge base the interface recognizes and resolves certain classes of ambiguity that arise from underspecified references in update requests. The most common issue is determining what database feature a given value refers to. The interface has three ways of resolving semantic ambiguity that may arise during the parse. These strategies use phrasal context, sentential context, and user consultation.

For example, consider the phrase "white patients." White could be a value in one or more databases that contain patient information, possibly with different meanings (racial origin versus current skin appearance, for instance). When this happens the interface first places white on the phrasal ambiguity list. Then it attaches the names of the fields where white is defined. If white has never been used before all the possible alphanumeric fields are attached. After parsing the entire phrase the interface attempts to resolve all ambiguities that arose. In this example the interface recognizes that the head noun is patients and retains only those possible field values of white that occur in the patients database. If this leaves only one possibility white may be resolved; otherwise it is passed upward and held for sentential disambiguation.
After the entire sentence has been parsed, if the ambiguity list is nonempty, the interface considers each value and possible field list on the ambiguity list in turn. The disambiguation process moves down by levels. On each level, the number of possible fields is reduced. At the first level the interface considers all possible values that were attached by the parser. At the second level, the fields that already appear in the IQL registers are removed from consideration. If at any of the above levels only one choice remains the user is consulted from that level. At the third level, the user is consulted to accept the interface’s interpretation of the ambiguous value or provide the correct interpretation. A pseudo-code algorithm follows.

```plaintext
level-one(possible-fields) {
    if (number-of(possible-fields) == 1) {
        11-field = consult(possible-fields);
    }
    else {
        11-field = level-two(possible-fields);
    }
    return(11-field);
}

level-two(possible-fields) {
    level2-flds = possible-fields - fields-in-IQL;
    if (number-of(level2-fields) == 1) {
        12-field = consult(level2-flds);
    }
    else {
        12-field = level-three(level2-flds);
        if (12-field == nil) {
            12-field = consult(possible-fields - level2-flds);
        }
    }
```
This multi-level consultation is designed to minimize user distraction. During the consultation, the interface presents the user with the ambiguous value followed by the possible meanings that remain. If the intended meaning is not in the first list, the user must choose other. This choice pops the user up one level of disambiguation. He is then presented with the possible fields from that level (minus those from lower levels). The user may choose other until he is at the top level, which contains all field names attached by the parser. Based on the example interactions shown here, the correct interpretation is found in either the first or second list presented to the user.

4.1.2. Consistency Maintenance

For database updates, the prototype system has implemented some very simple consistency maintenance checks. Rules governing legal field values may be entered in the knowledge base. Current rules allow for setting minimum and maximum values for numeric fields. After parsing an addition or
change, the knowledge base is examined to insure that the new information does not violate any rules. If a rule is violated, the user is informed of the violation and asked to enter a correct value. The interface does not check if this new value violates any rules, thus allowing the user to override the rules if needed.

4.2. Knowledge Base and Lexicon Maintenance

When an addition is parsed, the knowledge base must be updated. This includes adding any new non-numeric values to the knowledge base as well as adding new words to the lexicon. The same actions must be taken when a change is parsed. When a deletion is parsed, no changes must be made to either the lexicon or the knowledge base. Any values not appearing in the remaining database are kept for possible later use.

4.3. Intermediate Query Language

When an addition, is parsed the IQL registers are filled as follows.

Retrieval:
nil

Cualification:
(<filename> . <keyfieldname> = <keyfieldvalue>)

Operation:
ADD
Update:
  (<filename> . <fieldvalue> = <fieldvalue>)

Join:
  (<fromfile> . <fromfield> <tofile> . <tofield>)

Putting the new key value into the qualification register aids in response generation. The update register contains the information needed to fill the new record's fields. Finally, the join register contains the information needed for any necessary joins.

The system records changes in IQL much the same way as it does additions, except for the operation and qualification registers. The operation register contains CHANGE, while the qualification register describes the range of records to be changed. This description can include the connective AND and superlatives GREATEST and LEAST, as well as common arithmetic operations such as >, <=, etc.

After a deletion has been parsed, the retrieval and update registers are nil, the operation register contains DELETE, and the qualification and join registers are filled as in the change operation.

4.4. IQL Post-Processor

IQL was designed to be easily translated to many different DBMS languages. Safigan [Safigan 1987] developed a Model 204 translator for the unextended prototype; Diaz [Nutter,
Safigan, and Diaz 1989] developed a translator for McMax. Model 204 is a procedural DBMS language. That is, it was assumed that the user would program the database using the Model 204 language. The DBMS executes this program to carry out the user’s commands. McMax, on the other hand, is meant to be highly interactive. It runs under the Macintosh environment and supports such user friendly objects as menus and dialog boxes. Having the source code for both of these translators eased the development of a dBase III translator especially since McMax is similar to dBase. The dBase language was originally developed for early DOS machines and does not support windowed interactions, although separate workspaces are provided. The user is expected to program the database. The port to dBase III took about a week.

For the previously existing parts of the system, the port to dBase III was unremarkable. This report will, therefore, focus on the part of the post-processor that deals with new features (i.e., database update, response generation). A complete listing of the post-processor may be found in appendix C.

Like many other languages, dBase III contains commands for opening files, looping constructs, as well as mathematical operations and variables. It also provides for switching among ten currently open workspaces. It is through these workspaces that joins are implemented. The current
translator is limited to the use of two workspaces and therefore only two files may be referenced in an IQL command. Relaxing this restriction requires a slightly more complex post-processor, and appears to be more time consuming than difficult.

4.4.1. Initialization

At the start of the dBase program, all the variables are cleared, headings are turned off, and the display of record numbers is disabled. This is done so that the response generator can more easily access the actual data returned by the DBMS. The system is now ready to identify the active files and place them into appropriate workspaces. This is done by using the select and join commands.

The select command selects a workspace and assigns it a reference. In this post-processor implementation, integers are used as references. After a workspace is selected, the use command opens a database file and places it in the current workspace. For each unique filename found in the IQL command a select/use pair is generated of the form:

```
SELECT <workspace-number>
USE <file>
```
4.4.2. Additions

Next the translator examines the operation register. For an operation register value of ADD, the translator takes the value in the qualification register as the new key value and the values in the update register as the field values that correspond to this key. The following commands are generated:

APPEND BLANK
REPLACE <keyfield> WITH <value> ( .and. ;
    <field> WITH <value> )*  

The parentheses and asterisk are not generated; they are to imply grouping and the Kleene Star operation of zero or more repetitions. This group is repeated for as many values as exist in the update register.

4.4.3. Changes

If the operation is CHANGE, the translator must first examine the join register. If the join register is nil then no files must be joined and the operation is easy. The system first lists the field names and the corresponding field values from the update register. After all updated values are listed the qualification register is examined to determine the scope of these changes. The commands generated are as follows:
REPLACE ALL <field> WITH <value> (.and. ;
<field> WITH <value>)* ;
FOR <field> <comparator> <value> (.and. ;
<field> <comparator> <value>)*

If the join register is not empty then the root and the leaf of the join are determined from the join register. As discussed above, the structure of the join register is:

(<fromfile> . <fromfield> <tofile> . <tofield>)

This can be thought of as:

(<root-filename> . <root-fieldname>
<leaf-filename> . <leaf-fieldname>)

Since updates to the leaf of the join would produce possibly many side effects, usually unintended and unknown, only updates to the root of the join are allowed. A join requires that the translator find the workspace that each file is associated with. This is done by evaluating the filename. The filename was set with the workspace number it was assigned when workspaces were generated during the program initialization. The system selects the leaf file and qualifies it. Then the join field value for this subset is set to be equal to the system variable JOIN_VAR. The translator next turns to the root relation.

After the root relation is selected the system must choose to either use JOIN_VAR to qualify the root relation or to update the join field in tuples of the root relation. If
the reference to the leaf relation is contained in the qualification register then JOIN_VAR is used to qualify the root relation. The following commands are generated.

```
SELECT <eval leaf>
    LOCATE FOR <field> <comparator> <value> (.and. ;
    <field> <comparator> <value>)*
    JOIN_VAR = <joinfieldvalue>

SELECT <eval root>
    REPLACE ALL <field> WITH <value> (.and. ;
    <field> WITH <value>)* ;
    FOR <joinfield> = JOIN_VAR (.and. ;
    <field> <comparator> <value>)*
```

If the reference is in the update register then the join field in the tuples qualified by the qualification register is updated to be the value of JOIN_VAR. In the this case the commands generated are:

```
SELECT <eval leaf>
    LOCATE FOR <field> <comparator> <value> (.and. ;
    <field> <comparator> <value>)*
    JOIN_VAR = <joinfieldvalue>

SELECT <eval root>
    REPLACE ALL <joinfield> WITH JOIN_VAR (.and. ;
    <field> WITH <value>)* ;
    FOR <field> <comparator> <value> (.and. ;
    <field> <comparator> <value>)*
```

4.4.4. Deletions

The final update operation to be considered is DELETE. When the operation register is DELETE, the retrieval and update registers are nil. The join register contains join
information. First, consider the case when the join register is nil, i.e., we must only access one relation. The qualification register is examined and the following commands are generated:

```
DELETE ALL ;
  FOR <field> <comparator> <value> (and. ;
   <field> <comparator> <value>)*
```

If the join register is not nil, then the commands generated are as follows:

```
SELECT <eval leaf>
  LOCATE FOR <field> <comparator> <value> (.and. ;
   <field> <comparator> <value>)*
  JOIN_VAR = <joinfield>value

SELECT <eval root>
  DELETE ALL ;
   FOR <joinfield> = JOIN_VAR (.and. ;
     <field> <comparator> <value>)*
```

Note that it is impossible to delete a tuple from the leaf relation. If this were possible then we may have many references to that tuple with no target. To avoid dangling references we must either delete the referencing tuples or consult with the user to determine a new target. Both of these actions violate the rule of updating the database by causing the least amount of disruption.
4.5. Response Generation

The final action taken by the interface is response generation. The response is generated from the IQL registers. The response generator implemented here can respond to both database queries and database updates. A complete listing of the arcs and nodes of the response generator ATN can be found in appendix D. The sentence structure is very simple and general. For example, the system responds to additions and changes with

"I UNDERSTAND THAT <qualifications> [TO HAVE | TO BE (NOT)] A | AN] [<fieldname> OF <fieldvalue> {unit-of-measure} | <boolean fieldname>]."

Acknowledgement of a deletion is made of the form

"I have deleted <qualifications>."  

Other responses are similar. For example, the response to a count operation is

"There are <answer> <qualifications>."  

Some examples of responses the system may generate are:

I understand that Mary has a sex of female.  
I understand that Tom is not married.  
I have deleted cars that have a color of red.  
There are 10 patients that are married.  
Yes, there are patients that have a diagnosis of acute-mi.
Determining the subject of the response is done in two different ways. If the qualification register contains only a reference to a particular tuple through the key field, then this value is used as the subject of the response, and no other qualifications are needed. If there are more than this one qualification, another method must be used.

In order to determine what the subject of a response not involving a key value should be, the interface first examines the join register. If there is information in the join register the interface uses the filename, or synonym, if one exists, of the root of the join. If the join register is nil, then determining the subject is as easy as finding what file is being used. The interface chooses the filename from either the retrieval or qualification register.

The response generator qualifies the subject of the response in a very simple manner. It examines each field name in the qualification register in turn. For a boolean response the interface appends a form of to be to the response after the relative pronoun who. Next, if the boolean is false, a not is added before the fieldname otherwise just the fieldname is added to the response.

An alphanumeric or numeric field is handled slightly different. If the field is alphanumeric or numeric the
correct form of the verb to have is added before the proper
determiner is appended to the response (either a or an
depending on the initial letter of the fieldname). The
fieldname, or synonym, if one exists, is then added to the
end of the response. After the fieldname, the interface
appends the preposition of. Next the actual value is added.
In the case of a numeric field type the proper unit of
measure is found in the knowledge base and appended.

Adding the update to the response is done in the same manner
as qualifying the subject, except the update register is
used in place of the qualification register.

4.6. Novel Contributions of this Thesis

- Designed and implemented an update expert.
- Designed and implemented extensions to IQL to
  accommodate updates.
- Implemented a new IQL post-processor for
dBase III.
  - Third post-processor for TIPS.
  - Reproduces all features of past post-
    processors.
  - Handles new extensions to IQL.
  - First post-processor to manage updates.
- Designed and implemented first moves toward consistency and data integrity maintenance through interface.

- Designed and implemented a response generator.
5. FUTURE WORK

Future work may be divided into three categories: extensions, novel implementations, and theoretical work. Extensions are well understood and need only be added to the interface through brute force. Novel implementations are understood in broad terms but have never been implemented in the context of a fully transportable interface. Finally, theoretical work, including empirical studies, needs to be done to determine how to deal with some general issues. The results of this work can then be implemented in the interface.

5.1. Extensions

The most obvious extension to the system is enhanced parser coverage. Although the current system accepts a range of sentences, there are many limitations. Perhaps the most useful extension would be the ability to handle conjunctions and disjunctions. There are problems in dealing with these constructs, but the parser can do much better than it currently does without too much additional work. Another parser extension involves joins. Currently the system can only deal with one-to-one or many-to-one joins. Although many databases contain only joins of this type, extending the parser and including semantics to handle one-to-many and
many-to-many relations is desirable. The primary effort of this thesis has been to demonstrate the viability of the update and response architecture, not to achieve broad language coverage.

5.2. Novel Implementations

5.2.1. Enhanced Lexicon Management

The current system knows only words it has directly encountered, either in the interview or in processing requests. It would be interesting to see how adding a lexicon of words and their synonyms enhances the interface. For example, if the system knows of white as an adjective describing race then it should understand "Caucasian patients" as being "white patients" without the manager having to declare explicitly that Caucasian is a synonym for white. It would be particularly interesting to couple some version of TIPS with a pared-down version of a relational lexicon such as the one described in [Nutter 1990; Nutter, Fox and Evans 1990].
5.2.2. Error Handling

Error detection and correction is straightforward. This represents a major amount of work, but at the least, implementing an error detection and correction ability, such as that found on some compilers today [Aho, Sethi, and Ullman 1988], should be relatively straightforward.

5.2.3. Enhanced Rule Enforcement

Another enhancement is in the area of rule enforcement. Currently, the system accepts and enforces only rules restricting the values of numeric fields. Extending this idea to other fields seems easy. For example, a rule might be that the only valid values for sex are male and female. A more challenging rule extension deals with field relations.

Having the ability to define relations that must hold between two or more fields would be highly valued. For example, we might define a rule that states if a patient has prostate cancer then that patient must have a sex of male. This rule serves two purposes. First, it helps to maintain an error free database because any patient that is defined to have prostate cancer must be male. Second, it can serve as a useful inferencing tool. For example, if the above rule is present and if the system knows that a particular
patient has prostate cancer, then the system may automatically infer that such a patient has a sex of male. This has the potential of freeing the user from having to state many "obvious" facts.

At present, rules like these may be entered manually by a person familiar with SMePS inferencing. This "hard wiring" is unacceptable for a real system. Further work in this area should be directed toward allowing the system administrator to define these rules as natural language statements either during normal update operations or during the initial interview.

5.3. Theoretical Work

5.3.1. Multi-Valued Fields

Currently, the system is able to process only single valued fields. The interface can not process fields that may contain sub-fields. For example, a relation may be defined to have a multi-valued key field, say, supplier name and part number. There may be many parts supplied by the same supplier and many suppliers that supply the same part. By including both parts of the key in an update, one may, for example, describe the quantity of a specific part supplied
by a specific supplier. The current interface is incapable of processing this many-to-many relationship.

5.3.2. Joins

The entire issue of how to handle joins requires more study. One possible goal is minimizing user consultation. The current approach addresses this issue by minimizing side effects. On the other hand, one may not be willing to allow the database modifications that are required for such a blanket solution. For example, it may be illegal for certain users, say nurses, to change the diagnosis of a patient.

The choice made here, minimization of side effects, is not perfect. While minimizing user consultation, this choice requires that the join fields be modified. In some situations, this is unacceptable.

An alternative, user consultation, requires the user to specify the actions that must be taken. This approach forces the interface to consult repeatedly with the user about the detailed structure of the database. Furthermore, it presupposes that the user has at least enough knowledge to understand the database structure. One may argue that only a user that has detailed structural knowledge should be allowed to update the database. In some cases, one may even argue that updates through joins should be disallowed.
entirely. The choice to allow, restrict, or prohibit updates through joins depends on the semantics associated with the database.

Perhaps some sort of hybrid approach will work best. For example, say that certain preferences are identified during the interview process. One preference may be that a patient's diagnosis doesn't change often. Given this preference, the user should be consulted when an update is made that requires changing the diagnosis. On the other hand, perhaps a preference is that managers seldom change departments, or that they change departments less often than workers. Then, an update requiring a worker to change departments can be made without asking if either the worker or the manager should change departments.

The user consultation approach is slightly more complicated than the approach used here. First, the system must identify the ambiguity, which is done here. Second, the system must identify all possible updates that may be made. Given the knowledge stored in the knowledge base, this should not be too difficult. Finally, the system must present these to the user and ask for a choice. The user consultation would be similar to the approach taken here to disambiguate references.
5.3.3. Response Structure

Determining the amount of expansion that is needed deserves future study. For example, in some cases it may be enough to respond using the value of the key field, as in "I understand that Mary is married." In other cases a more exact subject may be desired. For example, "I have deleted patients that have a name of Mary." An empirical study to determine the level of expansion that actual users are comfortable with would be useful.

5.3.4. User Modeling

An effective natural language interface should be able to cooperate with and model the user. In everyday communication, people cooperate with each other a great deal. Some of this cooperation takes the form of helpful feedback.

When people interact with each other, they form a model of the knowledge that the other person has. If an inconsistency arises between what we think they know and what we know exists, we try to correct it. For example, if a person asks how many students passed economics last term and we know that economics wasn’t offered last term, we are generally helpful [Allen 1983]. Instead of responding "none", we probably respond with "Economics was not offered
last term." Another sort of cooperation refers to monopolizing conversation.

People use conservation of words to make sentences shorter and to provide the listener with more information about the answer. For example, if you ask "Who has company cars?" and I know that Bill, Mary, and Ted have company cars, I could enumerate each person as a reply. On the other hand, if I do some thinking, I might realize that these employees are also the vice-presidents of the company. Therefore, a more compact, and more informative, response is "All vice-presidents have company cars."

Clearly this form of cooperation involves a lot of processing. Some work in this area has been done for untransportable front-ends [Kaplan 1978; Kalita, Jones and McCalla 1986]. It remains a challenge to incorporate responses such as this into fully transportable interfaces.

User modeling is also helpful in disambiguation. If the interface can keep a model of the user’s view of the database, then it can use this model to determine the fields the user is referring to when he uses an ambiguous field value. For instance, suppose the interface knows that white may refer to two fields (race or skin appearance, as discussed earlier). Suppose that the interface has a model of the user. This model may include what fields he has been
referring to lately. Now, when white is used to describe something, the interface can make an informed choice of which field name the user is referring to.
6. CONCLUSIONS

Managing updates in a fully transportable interface requires constant interaction between the knowledge base and the other parts of the interface. A flexible and unified knowledge base is essential. A parser that can accept unknown words and make meaningful assumptions about the meanings of these words is also essential.

A successful update expert and response generator are developed here while maintaining two dimensions of transportability. The update expert uses the knowledge of the domain stored in the knowledge base and knowledge of words stored in the lexicon to parse an update request into an intermediate query language, IQL. The update expert recognizes certain classes of ambiguity and disambiguates with a minimum of user assistance. The update actions are checked to insure compliance with semantic constraints placed on the knowledge base by the system administrator. Control then passes to the post-processor and response expert.

The post-processor converts the IQL update into the target DBMS language. After the database operations have been performed, the response expert generates a natural language response from the IQL update and the results produced by the database operations.
The experiences here have shown that processing updates and generating responses for a full two dimensionally transportable interface is possible. The system developed here does not represent the ultimate answer for natural language information management. The hope is that a system such as this will give people not familiar with or comfortable with today’s commercial database management systems access to the wealth of information stored in them.
7. BIBLIOGRAPHY


APPENDIX A: Update Expert Grammar

The update expert accepts sentences from the following grammar.

\[
\begin{align*}
<\text{UPDATE}> & \rightarrow <\text{SUBJ}> <\text{S/NP}> \\
<\text{SUBJ}> & \rightarrow \{<\text{DET}>\} <\text{SUBJ}/\text{DET}> \\
<\text{SUBJ}/\text{DET}> & \rightarrow <\text{SUPERLATIVE}> <\text{SUBJ}/\text{DET}> \\
& \rightarrow <\text{PRONOUN}> <\text{SUBJ}/\text{HEAD}> \\
& \rightarrow \text{BOOLEAN-FIELD-NAME} <\text{SUBJ}/\text{DET}> \\
& \rightarrow \text{KEY-VALUE} <\text{SUBJ}/\text{HEAD}> \\
& \rightarrow \text{ALPHA-VALUE} <\text{SUBJ}/\text{DET}> \\
& \rightarrow \text{FILENAME} <\text{SUBJ}/\text{HEAD}> \\
& \rightarrow <\text{NUM-ADJ}> <\text{SUBJ}/\text{DET}> \\
<\text{SUBJ}/\text{HEAD}> & \rightarrow <\text{REL-CLAUSE}> \\
<\text{REL-CLAUSE}> & \rightarrow <\text{REL-PNOUN}> <\text{S/NP}> \\
<\text{REL-PNOUN}> & \rightarrow \text{RELATIVE PRONOUN AS DEFINED IN LEXICON} \\
<\text{S/NP}> & \rightarrow \text{TO HAVE} (<\text{DET}>\}) <\text{FIELDVALUE} \text{FIELDNAME} \\
& \rightarrow \text{REL-CLAUSE} \text{FIELDNAME} \text{OF} <\text{FIELDVALUE} <\text{REL-CLAUSE}> \\
& \rightarrow \text{TO BE} <\text{SUB/BE}> <\text{REL-CLAUSE}> \\
& \rightarrow <\text{SUBJ-FIELD}> <\text{REL-CLAUSE}> \\
& \rightarrow <\text{VERB-PHRASE}> <\text{REL-CLAUSE}> \\
<\text{SUB/BE}> & \rightarrow <\text{VERB-PHRASE}> \\
& \rightarrow \{\text{NOT}\} <\text{NP}> \\
& \rightarrow \{\text{NOT}\} <\text{PREDICATE}> \\
<\text{PREDICATE}> & \rightarrow \text{BOOLEAN-FIELD-NAME} \\
& \rightarrow <\text{NUM-ADJ}> \\
& \rightarrow \text{ALPHA-VALUE} \\
<\text{SUBJ-FIELD}> & \rightarrow \text{FIELDNAME TO BE FIELDVALUE} \\
<\text{NP}> & \rightarrow <\text{DET}> <\text{NP}/\text{DET}> \\
<\text{NP}/\text{DET}> & \rightarrow \text{BOOLEAN-FIELD-NAME} <\text{NP}/\text{DET}> \\
& \rightarrow \text{KEYVALUE} \\
& \rightarrow \text{ALPHA-VALUE} <\text{NP}/\text{DET}> \\
& \rightarrow \text{FILENAME} \\
& \rightarrow <\text{NUM-ADJ}> <\text{NP}/\text{DET}> \\
<\text{NUM-ADJ}> & \rightarrow \text{NUMBER UNITS-OF-MEASURE MAX-ADJ} \\
& \rightarrow \text{NUMBER UNITS-OF-MEASURE MIN-ADJ}
\end{align*}
\]
<VERB-PHASE>  \rightarrow  \text{VERB}  \{  <\text{DET}>  \}  \text{FIELDVALUE}  \\
              \rightarrow  \text{VERB}  \{  <\text{PREP}>  \}  \text{FIELDVALUE}  \\
<\text{DET}>  \rightarrow  \text{DETERMINER AS DEFINED IN LEXICON}  \\
<\text{PREP}>  \rightarrow  \text{PREPOSITION AS DEFINED IN LEXICON}
APPENDIX B: Permanent Lexicon

The permanent lexicon follows. Note, it is quite small and contains only very general words. Domain specific terms are added to this lexicon through the definewords function.

("THEIR") ((ctgy . pnoun))
("THEY") ((ctgy . pnoun))
("SHE") ((ctgy . pnoun))
("HE") ((ctgy . pnoun))
("IT") ((ctgy . pnoun))
("HER") ((ctgy . pnoun))
("HER") ((ctgy . pnoun))
("HAD") ((ctgy . v) (root . have) (tense . past))
("HAS") ((ctgy . v) (root . have) (tense . pres))
("HAVE") ((ctgy . v) (tense . pres))
("A") ((ctgy . det))
("AN") ((ctgy . det))
("THE") ((ctgy . det))
("OF") ((ctgy . prep))
("ABOUT") ((ctgy . prep))
("AT") ((ctgy . prep))
("FOR") ((ctgy . prep))
("IN") ((ctgy . prep))
("TO") ((ctgy . prep))
("WITH") ((ctgy . prep))
("BY") ((ctgy . prep))
("BE") ((ctgy . ident) (pres . is) (past . was) (pastp . been))
("IS") ((ctgy . ident) (root . be) (tense . pres))
("ARE") ((ctgy . ident) (root . be) (tense . pres))
("AM") ((ctgy . ident) (root . be) (tense . pres))
("WAS") ((ctgy . ident) (root . be) (tense . past))
("BEEN") ((ctgy . ident) (root . be) (tense . pastp))
("DO") ((ctgy . v) (pres . does) (past . did) (pastp . done))
"DOES" ((ctgy . v) (root . do) (tense . pres))
"DID" ((ctgy . v) (root . do) (tense . past))
"DONE" ((ctgy . v) (root . do) (tense . pastp))
"NUMBER" ((ctgy . quantity))
"AMOUNT" ((ctgy . quantity))
"QUANTITY" ((ctgy . quantity))
"WHAT" ((ctgy . wh-))
"WHERE" ((ctgy . wh-))
"WHEN" ((ctgy . wh-))
"WHO" ((ctgy . wh-)) ((ctgy . relclause))
"WHOSE" ((ctgy . relclause))
"WHICH" ((ctgy . relclause))
"THAT" ((ctgy . relclause))
"NO" ((ctgy . neg))
"NOT" ((ctgy . neg))
APPENDIX C: Knowledge Base Structures

The following structures comprise the knowledge base of the system (adapted from [Safigan 1987]):

1. Database filenames:

```
member
  ↓
<file-name>
```

2. Fieldnames:

```
member
  ↓
file
  ↓
<field-name>
```

3. Synonyms to filenames or fieldnames:

```
field
  ↓
synonym
  ↓
<field-name>
```

```
file
  ↓
synonym
  ↓
<file-name>
```
4. Key fields:

```
key
    ^
   / \
field
   |   
|   |
|   |
|   |
<field-name>  <filename>
```

5. A field may be alphanumerical, numeric, or Boolean. This is represented by one of the following structures:

```
member
    ^
   / \
Type
   |   
|   |
|   |
|   |
<field-name>  alphanumeric
```

```
member
    ^
   / \
Type
   |   
|   |
|   |
|   |
<field-name>  numeric
```

```
member
    ^
   / \
Type
   |   
|   |
|   |
|   |
<field-name>  Boolean
```

6. Numeric fields can have maximizing and minimizing adjectives. These are represented by:

```
max-adj
    ^
   / \
Field
   |   
|   |
|   |
|   |
<adjective>  <field-name>
```

```
min-adj
    ^
   / \
Field
   |   
|   |
|   |
|   |
<adjective>  <field-name>
```
7. Numeric fields have units of measure:

8. Numeric fields have upper and lower bounds:

9. Alphanumeric fields have associated values. These values are stored as:
10. A link between two relations is specified as:
APPENDIX D: Update Expert ATN Diagrams

The following pages are a graphical representation of the update expert. It is impossible to include all actions in the graph. Therefore, the graph should be examined in conjunction with the source code listing.
APPENDIX E: Update Expert Source Listing

The update expert makes use of the following functions.

```
(setq focus nil)
(setq my-hold nil)
(setq temp nil)
(setq filename-stack nil)
(setq hold-list nil)
(setq dis-hold nil)

;; adds a word to the sneps lexicon
;;
(defun definewords (word)
  (setq englex::undefs nil)
  (setq englex::*feature-values* nil)
  (englex::definewords word)
)

;; returns a register as a list - always
;;
(defmacro mygetr (reg)
  `(cond
      ;; ((listp (getr ,reg)) (getr ,reg))
      ;; (t (list (getr ,reg))) )
)

;; returns the nth element of lst
;;
(defun nthelem (num lst)
  (cond
    ;; ((null lst) nil)
    (((eql num 0) nil)
     (t (nth (1- num) lst)) )
  )
)

;; returns true if arg1 is greater than arg2 nil otherwise
;;
(defun greaterp (arg1 arg2)
  (cond
    ;; ((eql arg1 arg2) nil)
    (((eql (max arg1 arg2) arg1) t)
```
(t nil) )

; returns true if arg1 is less than arg2 nil otherwise
(defun lessp (arg1 arg2)
  (cond
    ((eql arg1 arg2) nil)
    ((eql (max arg2 arg1) arg2) t)
    (t nil) ))

; returns the name of the first file in register
(defun get-first-file (register)
  (cond
    ((eq (cdr register) '||) (car register))
    (t (get-first-file (cdr register)))))

; returns the first occurrence of referenced file in register
(defun g-ref (ref register)
  (cond
    ((null register) nil)
    ((null ref) nil)
    ((eq ref (car register)) (nthhead 5 register))
    ((and (member (car register) '(greatest least))
         (eq ref (cadr register))
      (nthhead 4 register))
    (t (g-ref ref (cdr register)))))

; returns list of all occurrences of referenced file in register
(defun get-refs (ref register)
  (cond
    ((null register) nil)
    ((null ref) nil)
    ((g-ref ref register)
     (and-insert (append (g-ref ref register)
                        (get-refs ref (d-ref ref register))))
    (t nil) )))

; insert AND between references in register
(defun and-insert (register)
  (cond
    ((null register) nil)
((member (car register) '(greatest least))
 (cond
   ((nthcdr 4 register)
      (append (nthhead 4 register)
        '(and)
        (and-insert (nthcdr 4 register)))
   (t (append (nthhead 4 register)
     (and-insert (nthcdr 4 register)))))
 ((eq (cadr register) '||))
 (cond
   ((nthcdr 5 register)
      (append (nthhead 5 register)
        '(and)
        (and-insert (nthcdr 5 register)) )
   (t (append (nthhead 5 register)
     (and-insert (nthcdr 5 register)))))
   (t (and-insert (cdr register))))) )

; deletes the first occurrence of referenced file from
; register
;
(defun d-ref (ref register)
 (cond
   ((null register) nil)
   ((null ref) register)
   ((eq ref (car register)) (nthcdr 6 register))
   (member (car register) '(and))
   (cond
     ((eq ref (cadr register)) (nthcdr 6 register))
     ((and (member (cadr register) '(greatest least))
        (eq ref (caddr register)) )
      (nthcdr 5 register) )
     (t (cons (car register)
       (d-ref ref (cadr register)))) )
   (t (cons (car register)
     (d-ref ref (cdr register)))) )
)

; delete all occurrences of ref file from register
;
(defun del-ref (ref register)
 (cond
   ((null register) nil)
   ((null ref) register)
   (member ref register)
   (del-refs ref (d-ref ref register))
   (t (cons (car register)
     (del-refs ref (cdr register)))) )
)
;;; returns the proper determiner to use for word
;;;
(defun get-det (word)
  (cond
   ((member (char (princ-to-string word) 0)
             "('#\a '#\e '#\i '#\o '#\u)) 'an)
     (t 'a))
  )

;;; returns true if value is between lower and upper
;;;
(defun between true if value is between lower and upper
  (cond
   ((and (lessp value upper) (greaterp value lower)) t)
     (t nil))
  )

;;; consult with user for a valid numeric value
;;;
(defun bounds-consult (fieldname fieldvalue lower upper)
  (princ "I'm sorry, ")
  (princ fieldvalue)
  (princ " is an unacceptable value for ")
  (princ fieldname)
  (princ ".")
  (line-feed)
  (princ "The values for ")
  (princ fieldname)
  (princ " must be between ")
  (princ lower)
  (princ " and ")
  (princ upper)
  (princ ".")
  (line-feed)
  (princ "Please enter the correct value for ")
  (princ fieldname)
  (princ " now: ")
  (read)
  )

;;; convert a character to a digit
;;;
(defun sym-to-digit (sym)
  (cond
   ((eq sym '#\0) 0)
   ((eq sym '#\1) 1)
   ((eq sym '#\2) 2)
   ((eq sym '#\3) 3)
   ((eq sym '#\4) 4)
   ((eq sym '#\5) 5)
((eq sym #\6) 6)
((eq sym #\7) 7)
((eq sym #\8) 8)
((eq sym #\9) 9)
(t 0) ) )

; help function to convert string to an integer
;(defun str-to-num (str count place)
  (cond
    ((eq count 0) 0)
    (t (plus (times place
      (sym-to-digit (char str (1- count))))
      (str-to-num str (1- count)
        (times 10 place)) )) ) )

; make an object into an integer
;(defun object-to-integer (obj)
  (str-to-num (princ-to-string obj)
    (array-dimension (princ-to-string obj) 0) 1))

; error handler
;(defun error-handler (error-condition wrd)
  (cond
    ((eq error-condition 'unknown-verb)
      (princ "I don’t understand the verb <")
      (princ wrd)
      (princ "> ! ")
      (line-feed)
      (echo 'abort-parse) ) ) )

; returns a list such that each element of lst occurs only
; once
; the order of lst is NOT preserved
;(defun singularize (lst)
  (cond
    ((null lst) nil)
    ((member (car lst) (cdr lst)) (singularize (cdr lst)))
    (t (cons (car lst) (singularize (cdr lst)))))

; remove references that are in qual register from retr
(define remove-dup-refs (retr qual)
  (cond
    ((null qual) retr)
    ((null retr) nil)
    ((member (nthelem 3 retr) qual)
      (remove-dup-refs (nthcdr 3 retr) qual))
    (t (append (nthhead 3 retr)
      (remove-dup-refs (nthcdr 3 retr) qual))))))

; delete each element of list1 from list2
(defun ldelete (list1 list2)
  (cond
    ((null list1) list2)
    ((null list2) nil)
    (t (ldelete (cdr list1) (delete (car list1) list2))))))

; find or define a word
(defun findordefine (word feature-list)
  (cond
    ((root-of word) t)
    (t (definewords (cons word feature-list)))))

; push wrd on the filename stack
(defun push-filename (wrd)
  (cond
    ((eq nil wrd) filename-stack)
    ((atom wrd) (setq filename-stack
      (merge (list wrd)
        filename-stack))
    ((listp wrd) (setq filename-stack
      (merge wrd filename-stack))))))

; return the top of filename stack
(defun list-filename () (car filename-stack))

; pop the filename stack
(defun pop-filename ()
  (setq temp (car filename-stack))
  (setq filename-stack (cdr filename-stack))
  (echo temp) )
(defun resolve (value field-list)
  (cond
    ((null field-list) nil)
    (t (terpri)
      (princ "By ")
      (princ value)
      (princ " are you refering to ")
      (terpri)
      (list-items field-list 1)
      (setq choice (read))
      (cond
      ((nthelem choice field-list)
          (nthelem choice field-list))
        (t nil) ) ) ) )

(print out a numbered list of items starting at num)
(defun list-items (lst num)
  (cond
    ((not (null lst)) (print-item num (car lst))
     (list-items (cdr lst) (+ num)))
    (null lst) (print-item num 'other) ) )

(print one item number num)
(defun print-item (num item)
  (princ "<")
  (princ num)
  (princ ">")
  (princ item)
  (terpri) )

(return a head of lst that is num elements long
returns nil if nul is 0
returns lst if either num == (len lst) or num > (len lst))
(defun nthhead (num lst)
  (cond
    ((eq num '0) nil)
    ((eq num (length lst)) lst)
    ((greaterp num (length lst)) lst)
    (t (reverse (nthcdr (diff (length lst) num)
      (reverse lst))))) ) )
; returns the arithmetic difference between n1 and n1
(defun diff (n1 n2)
  (system::* n1 n2)
)

; returns the arithmetic sum of n1 and n2
(defun plus (n1 n2)
  (system:+ n1 n2)
)

; returns the product of arg1 and arg2
(defun times (arg1 arg2)
  (system:* arg1 arg2)
)

; merge r1 and r2 together placing all superlatives from r1 at right end
; of r2. Assumes r2 is correct (has supers at right end)
(defun demote-superlatives (r1 r2)
  (cond
   ((null r1) r2)
   ((null r2) (cond
     ((not (member (car r1)
                  '(greatest least)))
      (demote-superlatives (nthcdr 5 r1)
                           (nthhead 5 r1)))
     (t (demote-superlatives
         (nthcdr 4 r1)
         (nthhead 4 r1)))
     ((eq 'and (car r1)) (demote-superlatives (cadr r1) r2))
     ((eq 'or (car r1)) (demote-superlatives (cadr r1) r2))
     ((eq 'not (car r1)) (demote-superlatives (cadr r1) r2))
     ((not (member (car r1) '(greatest least)))
      (demote-superlatives (nthcdr 5 r1)
                           (append (nthhead 5 r1)
                                    (cons 'and r2)))
     (t (demote-superlatives (nthcdr 4 r1)
                             (append r2
                                     (cons 'and
                                           (nthhead 4 r1)))))))))

(defun is-field (wrd)
  (cond
   ((find member wrd file ?x) (list wrd))
   (t nil))
)

(defun push-focus (keyvalue)
(setq focus (cons keyvalue focus))

(defun pop-focus ()
  (car focus))

; Merges two lists together, removing the duplicates.
;
; Programmer: Steve Safigan.
;
(defun merge (lst1 lst2)
  (cond
   ((null lst1) lst2)
   ((null lst2) lst1)
   ((member (car lst1) lst2) (merge (cdr lst1) lst2))
   (t (merge (cdr lst1) (cons (car lst1) lst2))))

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

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

; Programmer: Steve Safigan.
;
(defun locate-value (field lst)
  (cond
   ((eq nil lst) nil)
   ((eq field (caddr lst)) (nthelem 5 lst)))
(t (locate-value field (cdr 1st))) )

;; Programmer: Steve Saigan.
;;
(defvar remove-value (value 1st)
  (cond
    ((atom 1st) 1st)
    ((eq value (caddr 1st)) (nthcdr 5 1st))
    (t (append (nthhead 5 1st)
      (remove-value value (nthcdr 5 1st))))))

;; Programmer: Steve Saigan.
;;
(defvar find-values (value-list)
  (cond
    ((null value-list) nil)
    (t (append (cdadar value-list)
      (find-values (cdr value-list)))))))

;; Programmer: Steve Saigan.
;;
(defvar get-value (value-list data-value)
  (cond
    ((null value-list) nil)
    ((not (equal data-value (cdadar value-list)))
      (get-value (cdr value-list) data-value))
    (t (list
cdadar
      (find member (\(caar value-list\)) file ?x))
    )
    (caar value-list)
    '|
    (car data-value)))))))

;; Programmer: Steve Saigan.
;;
(defvar valid-digits-expressed-as-atoms
  (setq numbers
    '(['\|\|\|\|0|1|2|3|4|5|6|7|8|9|]))

;; Programmer: Steve Saigan.
;;
(defvar isnumber (str)
  (isnum str 0))

(defvar isnum (obj num)
(cond
  ((typep obj 'STRING) (setq str obj))
  ((typep obj 'SYMBOL) (setq str (string obj)))))
(cond
  ((equal (array-dimension str 0) num) t)
  ((member (char str num)
             '#\0 '#\1 '#\2 '#\3 '#\4 '#\5 '#\6 '#\7 '#\8 '#\9))
   (isnum str (1+ num)))
  (t nil))
)

; Programmer: Steve Saigan.
;
; returns the value passed to it
(defun echo (echoname) echnname)

; Programmer: Steve Saigan.
;
; reads the next value from the keyboard and converts it to an atom
(defun read-atom () (read))

; reads next value from keyboard and converts to a string
;mjb 12dec90
(defun read-string () (string (read)))

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

; Programmer: Steve Saigan.
;
; converts a list of atoms into one large atom name
(defun atomize (lst)
  (cond
   ((null lst) nil)
   ((atom lst) lst)
   ((null (cdr lst)) (car lst))
   (t (concatenate 'list
            (car lst)
            '\| |)
(atomize (cdr 1st)))))

;; Programmer: Steve Saigian.

;; atomizes list, then finds the root for a atom (if any) in the lexicon
(defun root-of (atm)
  (cond
    ((atom atm)
      (or (get-root
           (car (englex:lookup
                 (string-right-trim "EST" atm))))
        (get-root
          (car (englex:lookup (string atm))))
      ))
    ((stringp atm)
      (or (get-root
           (car (englex:lookup
                 (string-right-trim "EST" atm))))
        (get-root
          (car (englex:lookup atm)))
      ))
    ((equal (length atm) 1)
      (get-root (car (englex:lookup (atomize atm))))
    )
    (t
      (atomize (append
                 (reverse (cdr (reverse atm)))
                 (get-root
                   (car (englex:lookup
                         (car (reverse atm))))))))))

;; Programmer: Steve Saigian.

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

The update expert atn arcs follow.

;; start of a sentence

(s
  (jump s/ t
    (setq hold-list nil)
    (setq dis-hold-list nil)
    (setq dis-hold nil)
    (setq sentence-hold nil)
(setq filename-stack nil)
(setq phrase-hold nil)
(setq truthvalue 'true))

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

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

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

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

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

(wrd (delete) t
  (setq filename-stack nil)
  (setq truthvalue 'true)
  (to del/))

(push subj t
  (setr params *)
  (setr qualification
    (demote-superlatives (car (getr params))
      (getr qualification) )
    (setq phrase-hold (append (getr phrase-hold)
      (cadr (getr params)))
    (to subj/hold) )

  ; parse a deletion command
  ;
  (del/
    (push subj t (sendr truthvalue)
      (setr params *)))
(setr qualification (append (car (getr params)))
  (getr update)))
  (setq phrase-hold (cadr (getr params)))
  (setr operation 'delete)
  (to del/qual) ) )
(del/qual
  (jump del/qual (and (not (null phrase-hold))
    (not (null (car phrase-hold)))) )
  (setq sentence-hold
    (append sentence-hold
      (list (list 'qualification (car phrase-hold)))))
  (setq phrase-hold (cadr phrase-hold)) )
  (jump s/comp t) )
(subj/hold
  (jump subj/hold (and (not (null phrase-hold))
    (not (null (car phrase-hold)))) )
  (setq sentence-hold
    (append sentence-hold
      (list (list 'qualification (car phrase-hold)))))
  (setq phrase-hold (cadr phrase-hold)) )
  (jump s/pred t
    (setq dis-hold-list nil)
    (setq dis-hold nil)
    (setq hold-list nil) ) )
(s/pred
  (push s/np t
    (setr params *)
    (setr update (cond
      ((getr update)
        (append (getr update) '(and)))
        (t (getr update)) )
    (setr update (append (getr update)
      (car (getr params))))
    (setq phrase-hold (cadr (getr params)))
    (to s/pred/hold) ) )
(s/pred/hold
  (jump s/post-process/respond/done
    (eq (cadr (getr params)) 'error)
    (setr response (car (getr params))) )
  (jump s/pred/hold (and (not (null phrase-hold))
    (not (null (car phrase-hold)))) )
  (setq sentence-hold
    (append sentence-hold
      (list (list 'update (car phrase-hold))))))
  (setq phrase-hold (cadr phrase-hold))
  (jump s/comp t) )
; end of sentence, can be followed by another using AND
; Safigan
(s/comp
  (wrd (and) t
    (to s) )
  (push pp t
    (setr qualification (cond
      ((and (getr qualification)
        (getr *))
      (cons 'and
        (getr qualification)) )
    (setr qualification (append (getr *)
      (getr qualification)))
  (to s/comp) )
  (push s-disamb (not (null sentence-hold))
    (sendr reference (car sentence-hold))
    (sendr qualification)
    (sendr update)
    (setr params *)
    (setr qualification
      (descriptive-superlatives (car (getr params))
        (getr qualification) )
    (setr update (cond
      ((and (cadr (getr params))
        (getr update))
      (append (getr update) '(and)) )
    (setr update (append (getr update)
      (cadr (getr params))))
    (setq sentence-hold (cadr sentence-hold))
    (to s/comp) )
    (jump s/check-rules t
      (push-focus (getr qualification))
      (setr filenames
        (get-filenames (append (getr retrieval)
          (getr qualification)
          (getr update)) ))
    (setr join
      (cond
        ((null (cadr (getr filenames))) nil)
        ((find from-field-
          (find from-file (^(car (getr filenames)))
            to-file (^(cadr (getr filenames)))
            from-field ?x))
          (list
            (car (getr filenames))
            ',.|)
(car
  (find from-field-
    (find from-file (\(car (getr filenames))
      to-file (\(cadr (getr filenames))
      from-field ?x)))
    (cadr (getr filenames))
    |.|)
  (car
    (find to-field-
      (find from-file (\(car (getr filenames))
        to-file (\(cadr (getr filenames))
        to-field ?x)))
    ((find from-field-
      (find from-file (\(cadr (getr filenames))
        to-file (\(car (getr filenames))
        from-field ?x))
      (list
       (cadr (getr filenames))
       |.|)
    (car
      (find from-field-
        (find from-file (\(cadr (getr filenames))
          to-file (\(car (getr filenames))
          from-field ?x)))
      (car (getr filenames))
      |.|)
    (car
      (find to-field-
        (find from-file (\(cadr (getr filenames))
          to-file (\(car (getr filenames))
          to-field ?x)))
      (t '(error))
    (setr retrieval
      (remove-dup-refs (getr retrieval)
        (getr qualification))))
  (s/check-rules
    (push check-rules t
      (sendr retrieval) (sendr qualification)
      (sendr operation) (sendr update)
      (sendr join)
      (to s/post-process)))
  (s/post-process
    (jump s/respond t
      (post-process (getr retrieval) (getr qualification)
        (getr operation) (getr update)
        (getr join))))
  (s/respond
    (push respond t
      (sendr retrieval) (sendr qualification)
(sendr operation) (sendr update) (sendr join)
(setr response *)
(to s/post-process/respond) )
)

(s/post-process/respond
(to (s/post-process/respond/done) t) )

(s/post-process/respond/done
(pop response t) )

;
; disambiguate reference,
; reference = (register (value (possible-fields)))
; needs sendr’d reference, qualification, and update
;
(s-disamb
(jump sent-disamb t
(setr register (car (getr reference)))
(setr value (caadr (getr reference)))
(setr possible-fields (cadadr (getr reference))) ) )

(sent-disamb
; if more than one possible field disambiguate
(push many-possible-fields
(greaterp (length (getr possible-fields)) 1)
(sendr possible-fields) (sendr value)
(sendr qualification) (sendr update)
(setr field *)
(to sent-disamb/resolve) )
(jump sent-disamb/resolve t
(setr field (car (getr possible-fields))) ) )

(sent-disamb/resolve
(jump sent-disamb/build t
(eval (buildq (assert field + value +) field value))
(findordefine (getr value) '(((ctgy . t))))
(setr file (car (eval
(buildq (find file- (find member + file ?x))
field)))) ) )

(sent-disamb/build
(jump sent-disamb/done (eq (getr register) 'update)
(setr umods (append (getr umods) (list
(getr file)
'\.|
(getr field)
'= (getr value) )))) )

(jump sent-disamb/done (eq (getr register)
'qualification)
(setr qmods (append (getr qmods) (list
'(\|)
(getr file)
'= (getr value) )))))


109 (getr file)
'/.|
(getr field)
'= (getr value) ))) )))

(sent-disamb/done
  (pop (list (getr qmods) (getr umods)) t) )

(many-possible-fields
 ; get all known fields and remove used fields from
 ; possible field list
 (jump many/fields t
   (setr all-fields
     (eval (buildq
       (find member- (find member ?x file ?y)))))))
 (setr used-fields
   (get-fieldnames (append (getr qualification)
                         (getr update) ))))
 (setr possible-fields
   (idelete (mygetr used-fields)
            (mygetr possible-fields) ))
 (setr level1-fields (getr possible-fields) ) )

(many/fields
 (push disamb2 t
   (sendr possible-fields) (sendr value)
   (sendr qualification) (sendr update)
   (setr level2field *)
   (to many/level2) ) )

(many/level2
 (jump many/done (getr level2field)
   (setr fieldname (getr level2field)) )
 (jump many/done (nullr level2field)
   (setr fieldname
     (resolve (getr value)
              (idelete (mygetr level1-fields)
                       (mygetr all-fields) ))) )

(many/done
 (pop fieldname t) )

(disamb2
 ; get referenced files already in IQL
 (jump disamb2/files t
   (setr filenames
     (get-filenames (append (getr qualification)
                          (getr update) ))))) )
(disamb2/files
 ; remove fields of non-referenced files from
 consideration
 (push get-field-names t
  (sendr filenames)
  (setr ref-file-fields *)
  (setr orig-poss-fields (getr possible-fields))
  (setr possible-fields
   (overlap (getr ref-file-fields)
    (getr orig-poss-fields) ))
  (to disamb2/files) ) )

(disamb2/files
 (push resolve t
  (sendr possible-fields) (sendr value)
  (setr resolvefield *)
  (to disamb2/resolve) ) )

(disamb2/resolve
 (jump disamb2/done (getr resolvefield)
  (setr fieldname (getr resolvefield)) )
 (jump disamb2/done (nullr resolvefield)
  (setr fieldname
   (resolve (getr value)
    (ldelete (mygetr ref-file-fields)
     (mygetr orig-poss-fields))))))

(disamb2/done
 (pop fieldname t) )

(resolve
 (jump resolve/done t
  (setr fieldname (resolve (getr value)
   (mygetr possible-fields)))))

(resolve/done
 (pop fieldname t) )

;
; return the fieldnames associated with a list of filenames
; needs sendr’d filenames
;
(get-field-names
 (jump get-field-names (not (nullr filenames))
  (setr fieldnames
   (append (getr fieldnames)
    (eval (buildq
     (find member- (find member ?x file +))
     (car (mygetr filenames))))))
  (setr filenames (cdr (mygetr filenames)))) )
(jump get-field-names/done (nullr filenames)) }

(get-field-names/done
 (jump get-field-names/done (nullr filenames))
 (setr filenames
   (eval (buildq
       (find member- (find member ?x file ?z)))
   (pop filenames t)) )

; return the filenames associated with a list of fieldnames
; needs sendr'ed fieldnames
;
(get-file-names
 (jump get-file-names (not (nullr filenames))
   (setr filenames
     (append (getr filenames)
     (eval (buildq (find member + file ?x)
       (car (getr filenames)))))
     (setr filenames (cdr (getr filenames)))
   (jump get-file-names/done (nullr filenames)
     (setr filenames (getr filenames)))))

(get-file-names/done
 (jump get-file-names/done (nullr filenames))
 (setr filenames
   (eval (buildq
       (find file- (find member ?x file ?z)))
   (pop filenames t)) )

; current item is a pronoun and a subject is in focus
;
(pnoun
   (cat pnoun (setr subj (pop-focus))
     (to pnoun/end)) )

(pnoun/end
   (pop subj t))

;
; superlative form of maximizing adjective
;
(mymax-adj
   (to (mymax-adj/end)
     (and (root-of (getr *))
       (eval (buildq (find max-adj + field ?x)
         (root-of (getr *))) )}))
(setq fieldname
  (car (eval (buildq
      (find field- (find max-adj + field ?x))
      (root-of (getr *)) )))) )

(mymax-adj/end
  (pop fieldname t) )

;; superlative form of minimizing adjective
;;
(mymin-adj
  (to (mymin-adj/end)
    (and (root-of (getr *))
      (eval (buildq (find min-adj + field ?x)
                      (root-of (getr *)) )))
  (setq fieldname
    (car (eval
      (buildq (find field- (find min-adj + field ?x))
                      (root-of (getr *)) )))) )

(mymin-adj/end
  (pop fieldname t) )

;; start of subject noun phrase
;;
(subj
  (cat det t
    (to subj/det) )
  (jump subj/det t) )

(subj/det
  ;; superlative form of maximizing adjective
  (push mymax-adj t
    (setq operation 'change)
    (liftr operation)
    (setq filename *)
    (setq fieldname (car (eval (buildq
      (find file- (find member + file ?x))
      filename)))))
  (setq qmods (cond
      ((getr qmods)
        (append (getr qmods) '(and)))
      (t (getr qmods)) )
    (setq qmods (append (getr qmods) (list 'greatest
      (getr filename)
      '|' (getr fieldname) )))))
(successverb subj/det)

; superlative form of minimizing adjective
(push mymin-adj t
  (setr operation 'change)
  (liftr operation)
  (setr fieldname *)
  (setr filename (car (eval (buildq
                        (find file- (find member + file ?x))
                        filename))))
  (setr qmods (cond
                ((getr qmods)
                 (append (getr qmods) '(and)))
                (t (getr qmods)) )
  (setr qmods (append (getr qmods) (list
                        'least
                        (getr filename)
                        '|.
                        (getr fieldname)))))
  (to subj/det)
)

; pronoun
(push pronoun t
  (setr params *)
  (setr operation 'change)
  (liftr operation)
  (setr qmods (cond
                ((getr qmods)
                 (append '(and) (getr qmods)))
                (t (getr qmods)) )
  (setr qmods (append (getr qmods) (getr params)))
  (to subj/head)
)

; boolean field
(push boolean t (sendr truthvalue)
  (setr operation 'change)
  (liftr operation)
  (setr fieldname (car (getr *)))
  (setr fieldvalue (cadr (getr *)))
  (setr filename (car (eval (buildq
                        (find file- (find member + file ?x))
                        filename))))
  (setr qmods (cond
                ((getr qmods)
                 (append '(and) (getr qmods)))
                (t (getr qmods)) )
  (setr qmods (append (list
                        (getr filename)
                        '|.
                        (getr fieldname)
                        '='
                        (getr fieldvalue) )
(to subj/det))

; a known key value
(push np/key t
  (setr operation 'change)
  (liftr operation)
  (setr fieldname (car (getr *)))
  (setr fieldvalue (cadr (getr *)))
  (setr filename (car (eval (buildq
                      (find file- (find member + file ?x))
                      filename))))
  (setr qmods (cond
    ((getr qmods)
     (append '(and) (getr qmods)))
    (t (getr qmods))))
  (setr qmods (append (list
    (getr filename)
    '|' .
    (getr fieldname)
    '=
    (getr fieldvalue))
    (getr qmods)))
  (to subj/head))

; any known field value
(push get-value t
  (setr operation 'change)
  (liftr operation)
  (setr params *)
  (setr fieldname (car (getr params)))
  (setr fieldvalue (cadr (getr params)))
  (setr held (caddr (getr params)))
  (setq hold-list (append hold-list (getr held)))
  (to known-value))

; name of the file
(push np/filename t
  (push-filename (getr *))
  (setr subj-filename *)
  (setr operation 'change)
  (liftr operation)
  (to subj/head))

; numeric adjective, i.e. ... 20 year old ...
(push num-adj t
  (setr operation 'change)
  (liftr operation)
  (setr fieldname (car (getr *)))
  (setr fieldvalue (cadr (getr *)))
  (setr filename (car (eval (buildq
   (find file- (find member + file ?x))
   filename))))
  (setr qmods (cond
    ((getr qmods)
     (append '(and) (getr qmods)))
    (t (getr qmods))))
  (setr qmods (append (list
    (getr filename)
    '|' .
    (getr fieldname)
    '=
    (getr fieldvalue))
    (getr qmods)))
  (to subj/head))

; a known key value
(find file- (find member + file ?x))
  (fieldname))

(setr qmods (cond
  ((getr qmods)
    (append '(and) (getr qmods)))
  (t (getr qmods)) )

(setr qmods (append (list
  (getr filename)
    '/'
  (getr fieldname)
    '='
  (getr fieldvalue) )
  (getr qmods) )

(to subj/det) )

; new key value
(push new-key t
  (setr operation 'add)
  (liftr operation)
  (setq dis-hold-list
    (list (append hold-list (getr *))))
  (to subj/relclause) )

(subj/head
  (push relclause t
    (setr params *)
    (setr qmods
      (demote-superlative (car (getr params))
        (getr qmods)))
      (setq hold-list
        (append (getr held) (cadr (getr params)))))
      (setq dis-hold-list (list hold-list))
    (to subj/relclause) )
  (push pp t
    (setr params (getr *))
    (setr qmods (cond
      ((and (getr qmods) (getr *))
        (cons 'and (getr qmods)) )
      (t (getr qmods)) )
    (setr qmods
      (demote-superlative (car (getr params))
        (getr qmods)))
      (setq dis-hold-list (list hold-list))
    (to subj/relclause) )
  (jump subj/relclause t
    (setq dis-hold-list (list hold-list)) ) )

(subj/relclause
  (push disamb (not (null dis-hold-list))
    (sendr subj-filename)
    (setr params *)
(setq qmods
    (demote-superlatives (car (getr params))
    (getr qmods))
  (setq hold-list (append hold-list
    (cadr (getr params))))
  (to subj/end )
  (jump subj/end (null dis-hold-list) )

(subj/end
  (pop (list (getr qmods) hold-list) t) )

(jump subj/det (nullr held)
  (setq filename
    (car (eval (buildq
      (find file- (find member + file ?x))
      (findname)))))
  (setq qmods (cond
    ((getr qmods) (append '('and)
      (getr qmods)) )
  (setq qmods (append (list
    (getr filename)
    '|
    (getr fieldname)
    '=
    (getr fieldvalue) )
    (getr qmods) )))
(jump subj/det (not (nullr held))) )

(jump disamb/mods
  (and (not (null dis-hold-list))
    (eq (length (cadar dis-hold-list)) '1) )
  (setq filename (caadar dis-hold-list))
  (setq value (caar dis-hold-list))
  (setq filename
    (car (eval (buildq
      (find file- (find member + file ?x))
      (findname))))
  (setq dis-hold-list (cdr dis-hold-list)) )
(jump disamb-subj
  (and (not (null dis-hold-list))
    (not (eq (length (cadar dis-hold-list)) '1)) )))
(jump disamb/done (null dis-hold-list) )

(jump disamb (nullr subj-filename)
  (setq dis-hold (append dis-hold (car dis-hold-list)))
  (setq dis-hold-list (cdr dis-hold-list) )

(push get-field-names
  (and (not (nullr subj-filename))
       (setr filenames (list (getr subj-filename)))))
(setr subj-fieldnames *)
(setr fieldname
  (overlap subj-fieldnames
    (cadar (getr dis-hold-list))))
(to disamb-subj/field) )

(disamb-subj/field
  (jump disamb t
    (setq dis-hold (append dis-hold (car dis-hold-list)))
    (setq dis-hold-list (cdr dis-hold-list)) ) )

(disamb/done
  (pop (list (getr mods) dis-hold) t) )

;
; get a new key value, succeeds iff next word is a verb or
; fieldname
;
(new-key
  (to (new-key/value) t
    (setr key *)) )

(new-key/value
  (push verb/ t
    (setr params *)
    (setr fieldname (car (getr params)))
    (setr fieldname (cadr (getr params)))
    (push-filename (getr filename))
    (to new-key/file) )
(jump new-key/file
  (eval (buildq (find member + file ?x) *))
  (setq fieldname *)
  (setq filename
    (car (eval (buildq
       (find file- (find member + file ?x)) *)) )
    (push-filename (getr filename)) ) )

(new-key/file
  (jump new-key/done (nullr filename)
    (setq keyfields
      (eval (buildq
        (find key- (find key ?x file ?z))))) )
  (jump new-key/done (not (nullr filename))
    (setq keyfields
      (eval (buildq (find key- (find key ?x file +))
        filename)) ) )

(new-key/done
(pop (list (getr key) (getr keyfields)) t) )

; get a known field value or synonym return value and possible fields as held if ambiguous
(g-v/value
  (to (g-v/value)
    (eval (buildq (find field ?x value +) *))
    (setr fieldnames
      (eval (buildq
        (find field- (find field ?x value +)) *))
      (setr fieldvalue *))
    (to (g-v/value)
      (eval (buildq (find value ?x synonym +) *))
      (setr fieldvalue
        (eval (buildq
          (find value- (find value ?x synonym +)) *))
          (setr fieldnames
            (eval (buildq
              (find field- (find field ?x value +))
              fieldvalue))))
    )))

(g-v/value
  (jump get-value/end
    (and (eq 1 (length (getr fieldnames)))
      (not (eval "(buildq
          (find key + file ?x)
          (getr fieldnames))"))
      (setr held nil)
      (setr fieldname (car (getr fieldnames)))
      (push get-file-names
        (not (eq 1 (length (getr fieldnames)))
        (sendr fieldnames)
        (push-filename (getr *))
        (setr held (list (getr fieldvalue) (getr fieldnames)))
        (setr fieldname nil)
        (setr fieldvalue nil)
        (to get-value/end)) )

(get-value/end
  (pop (list (getr fieldname)
    (getr fieldvalue) (getr held)) t) )

(relclause
  (cat relclause t (to relclause/intro)) )

(relclause/intro
  (push s/npt
(setq params *)
(setq qmods
   (cond
    ((getq qmods) (append '(and) (getq qmods)))
    (t (getq qmods)))
   (setq qmods (append (car (getq params)) (getq qmods)))
   (setq hold-list (append hold-list
    (cadr (getq params))))
   (to relclause/end) ))

(relclause/end
   (pop (list (getq qmods) hold-list) t) )

; can have to have, to be, trans verb, or fieldname
; following subject
; (s/np

(push subj/has (string-equal (root-of (getq *)) "have")
   (setq params *)
   (setq umods
     (cond
      ((getq umods) (append (getq umods) '(and)))
      (t (getq umods)))
     (setq umods (append (getq umods) (car (getq params))))
     (setq hold-list (cadr (getq params)))
     (to s/np/end) ))

(push sub/be (string-equal (root-of (getq *)) "be")
   (setq params *)
   (setq umods
     (cond
      ((getq umods) (append (getq umods) '(and)))
      (t (getq umods)))
     (setq umods (append (getq umods) (car (getq params))))
     (setq hold-list (cadr (getq params)))
     (to s/np/end) ))

(push subj/field t
   (setq params *)
   (setq umods
     (cond
      ((getq umods) (append (getq umods) '(and)))
      (t (getq umods)))
     (setq umods (append (getq umods) (car (getq params))))
     (to s/np/end) ))

(push vp/ t
   (setq params *)
   (setq umods
(cond 
  ((getr umods) (append (getr umods) '(and))) 
  (t (getr umods))))
(setr umods (append (getr umods) (car (getr params))))
(to s/np/end) )

(push parse-error (setr error-condition 'unknown-verb)
  (sendr error-condition)
  (setr umods *)
  (setq hold-list 'error)
  (to s/np/end) ) )

(s/np/end
  (push relclause t
    (setr params *)
    (setr umods
      (cond
        ((getr umods) (append (getr umods) '(and)))
        (t (getr umods))))
    (setr umods (append (getr umods) (car (getr params))))
    (to s/np/end))
  (pop (list (getr umods) hold-list) t) ) )

(sub/be
  (to (sub/be/) (string-equal (root-of (getr *)) "be")
    (setq be-hold-list nil) ) )

(sub/be/
  (push vp/ t
    (setr params *)
    (setr umods
      (cond
        ((getr umods) (append (getr umods) '(and)))
        (t (getr umods))))
      (setr umods (append (getr umods) (car (getr params))))
      (to sub/be/end))
  (push subj/be t
    (setr params *)
    (setr umods
      (cond
        ((getr umods) (append (getr umods) '(and)))
        (t (getr umods))))
      (setr umods (append (getr umods) (car (getr params))))
      (setq be-hold-list (append be-hold-list
          (cadr (getr params))))
      (to sub/be/end))

  (sub/be/end
    (pop (list (getr umods) be-hold-list) t) ) )
(expr fieldname t
(setr fieldname *)
(setr filename
  (car (eval (buildq
      (find file- (find member + file ?x))
        (fieldname))))
  (to subj/field/s) ))

(to (subj/field/be)
  (string-equal (root-of (getr *)) "be")) )

(push numeric t
  (setr fieldvalue *)
  (to subj/field/be/value) )

(push get-value t
  (setr params *)
  (setr fieldvalue (cadr (getr params)))
  (to subj/field/be/value) )

(eval (buildq
      (assert field + value +) fieldname fieldvalue))

(findordefine (getr fieldvalue) '((ctgy . t)))) )

(jump subj/field/be/value/end t
 (setr umods
   (cond
     ((getr umods) (append (getr umods) '(and)))
     (t (getr umods)) )
   (setr umods (append (getr umods) (list
     (getr filename)
     '|
     (getr fieldname)
     '='
     (getr fieldvalue) )))))) )}
(subj/field/be/value/end
  (pop (list (getr umods)) t) )

;
; subject IS (check for negation)
;
(subj/be
  (push getneg t
    (setr truthvalue *)
    (setq is-hold-list nil)
    (to subj/be/neg) )
)

;
; <subject> IS {NOT}
;
(subj/be/neg
  (push np/
    (member (getr *) '("A" "AN") :test 'string-equal)
    (sendr truthvalue)
    (setr params *)
    (setr umods
      (cond
        ((getr umods) (append (getr umods) '(and)))
        (t (getr umods))) )
    (setr umods
      (append (getr umods) (car (getr params))))
    (setq is-hold-list
      (append is-hold-list (cadr (getr params))))
    (to subj/be/end) )
  (push pred/ t (sendr truthvalue)
    (setr params *)
    (setr umods
      (cond
        ((getr umods) (append (getr umods) '(and)))
        (t (getr umods))) )
    (setr umods
      (append (getr umods) (car (getr params))))
    (setq is-hold-list
      (append is-hold-list (cadr (getr params))))
    (to subj/be/end) )
)

(subj/be/end
  (pop (list (getr umods) is-hold-list) t) )

(pred/ {jump pred/hold t (setq pred-hold-list nil)} )

(pred/hold
  ; boolean field
(push boolean t (sendr truthvalue)
  (setr filename (car (getr *)))
  (setr fieldvalue (cadr (getr *)))
  (setr filename
      (car (eval (buildq
                    (find file- (find member + file ?x))
                    filename))))))

(setr umods
  (cond
   ((getr umods) (append (getr umods) ’(and!)))
   (t (getr umods)) ))
(setr umods (append (getr umods) (list
  (getr filename)
  ’|.
  (getr fieldname)
  ’=
  (getr fieldvalue) )))

(to pred/end) )

; numeric adjective, i.e. ... 20 year old ... (push num-adj t
  (setr filename (car (getr *)))
  (setr fieldvalue (cadr (getr *)))
  (setr filename
      (car (eval (buildq
                    (find file- (find member + file ?x))
                    filename))))

(setr umods
  (cond
   ((getr umods) (append (getr umods) ’(and!)))
   (t (getr umods)) ))
(setr umods (append (getr umods) (list
  (getr filename)
  ’|.
  (getr fieldname)
  ’=
  (getr fieldvalue) )))

(to pred/end) )

; any known field value
(push get-value t
  (setr params *)
  (setr filename (car (getr params)))
  (setr fieldvalue (cadr (getr params)))
  (setr held (caddr (getr params)))
  (setq pred-hold-list
    (append pred-hold-list (getr held)))
  (to pred-known-value) )

; a new field value put it on the hold-list to return
(push new-value
  (not (eval (buildq (find field ?x value +) *))))
(setq pred-hold-list
  (append pred-hold-list (list (getr *)))))
(to pred/end) )

(pred-known-value
  (jump pred/end (nullr held)
    (setr filename (car (eval (buildq
      (find file- (find member + file ?x))
        filename)))
    (setr umods
      (cond
        ((getr umods) (append '(and) (getr umods))
          (t (getr umods)) ) )
      (setr umods (append (list
        (getr filename)
        ','|
        (getr fieldname)
        ','=
        (getr fieldvalue) )
        (getr umods) )))
    (jump pred/end (not (nullr held)))) )

(pred/end
  (pop (list (getr umods) pred-hold-list) t) )

; ; look for negation
; (getneg
  (cat neg t
    (setr truthvalue 'false)
    (to getneg/end) )
  (jump getneg/end t
    (setr truthvalue 'true) ) )

(getneg/end
  (pop (getr truthvalue) t) )

; ; <subj> HAS <det> <fieldvalue> <fieldname>
; ;     <det> <fieldname> OF <fieldvalue>
; ; (subj/has
  (to (subj/has/v)
    (string-equal (root-of (getr *)) "have") ) )
(subj/has/v
 (cat det t (to subj/has/det))
 (jump subj/has/det t) )

(subj/has/det
 ; filename
 (push filename t
 (setr filename *)
 (setr filename
 (car (eval
 (buildq (find file-
 (find member + file ?x)) *)) )
 )
 )
 (to subj/has/det/fieldname) )
 ; any known field value
 (push get-value t
 (setr params *)
 (setr fieldvalue (cadr (getr params)))
 (to subj/has/det/fieldvalue) )
 ; assume its a fieldvalue
 (to (subj/has/det/fieldvalue) t
 (setr fieldvalue *)) )

(subj/has/det/fieldvalue
 ; filename
 (push filename t
 (setr filename *)
 (eval (buildq
 (assert value + field +) fieldvalue filename))
 (findordefine (getr fieldvalue) ’(((ctgy . t))))
 (setr filename
 (car (eval
 (buildq (find file-
 (find member + file ?x)) *)) ))
 )
 (to subj/has/done) )

(subj/has/det/fieldname
 (wrd (of) t (to subj/has/det/fieldname/of)) )

(subj/has/det/fieldname/of
 (push numeric t
 (setr fieldvalue *)
 (to subj/has/done) )
 ; any known field value
 (push get-value t
 (setr params *)
 (setr fieldvalue (cadr (getr params)))
 (eval (buildq (assert value + field +)
 fieldvalue filename))
 (to subj/has/done) )
 (to (subj/has/done) t
 (setr fieldvalue *)
 (eval (buildq (assert value + field +)
(subj/has/done
 (jump subj/has/pop t
  (setr umods
   (cond
     ((getr umods) (append (getr umods) '(and)))
     (t (getr umods))) )
   (setr umods (append (getr umods) (list
     (getr filename)
     '|' |
     (getr fieldname)
     '='
     (getr fieldvalue))))) )
)
(subj/has/pop
 (pop (list (getr umods) nil) t) )

(np/ (cat det t (to np/det))
 (jump np/det t) )

(np/det

 ; boolean field
 (push boolean t (sendr truthvalue)
  (setr fieldname (car (getr *))
  (setr fieldvalue (cadr (getr *))
  (setr filename
   (car (eval (buildq
     (find file- (find member + file ?x))
     filename)))))
  (setr umods
   (cond
     ((getr umods) (append (getr umods) '(and)))
     (t (getr umods)) )
   (setr umods (append (getr umods) (list
     (getr filename)
     '|' |
     (getr fieldname)
     '='
     (getr fieldvalue)))))
  (to np/det))
; a key value
(push np/key t
  (setf fieldname (car (getr *)))
  (setf fieldvalue (cadr (getr *)))
  (setf filename
   (car (eval (buildq
                (find file- (find member + file ?x))
                filename)))))
  (setf umods
   (cond
    ((getr umods) (append (getr umods) '(and)))
     (t (getr umods)) )))
  (setf umods (append (getr umods) (list
                             (getr filename)
                             '||
                             (getr fieldname)
                             '='
                             (getr fieldvalue) )))
 (setq dis-hold-list (list hold-list))
 (to np/head))

; any known field value
(push get-value t
  (setf params *)
  (setf fieldname (car (getr params)))
  (setf fieldvalue (cadr (getr params)))
  (setf held (caddr (getr params)))
  (setq hold-list (append hold-list (getr held)))
  (to np-known-value))

; name of the file
(push np/filename t
  (setq dis-hold-list (list hold-list))
  (to np/head))

; numeric adjective, i.e. . . . . 20 year old . . .
(push num-adj t
  (setf fieldname (car (getr *)))
  (setf fieldvalue (cadr (getr *)))
  (setf filename
   (car (eval (buildq
                (find file- (find member + file ?x))
                filename)))))
  (setf umods
   (cond
    ((getr umods) (append (getr umods) '(and)))
     (t (getr umods)) )))
  (setf umods (append (getr umods) (list
                             (getr filename)
                             '||
                             (getr fieldvalue) )))
  (to np/head))
(getr fieldname)
(\'= 
(getr fieldvalue) )

(to np/det) )

; a new field value
(push new-value
  (not (eval (buildq (find field ?x value +) *)))
  (setq hold-list (append hold-list (list (getr *)))))
(to np/det) )

; disambiguate as much of the hold-list as possible
;
(np/head
  (push disamb (not (null dis-hold-list)))
  (setr params *)
  (setr umods (append (getr umods) (car (getr params))))
  (setq hold-list (cadr (getr params)))
  (to np/end) )
(jump np/end (null dis-hold-list) )

(np-known-value
  (jump np/det (nullr held)
    (setr filename
      (car (eval (buildq
        (find file- (find member + file ?x))
        (findname)))))
      (setq umods
        (cond
          ((getr umods) (append ''(and) (getr umods)))
          (t (getr umods)) )
        (setq umods (append (list
          (getr filename)
          '\.'
          (getr fieldname)
          '//'
          (getr fieldvalue) )
          (getr umods) ))) )

(jump np/det (not (nullr held)) )

(np/end
  (pop (list (getr umods) hold-list) t) )

; returns the value and possible fieldnames
;
(new-value
  (to (new-value/done) t
    (setr value *)
    (setr fieldnames

(eval (buildq
    (find member-  
    (find member ?x type alphanumeric)) )) 
)

; remove the key field names
(setr fieldnames
    (delete (eval (buildq
        (find key- (find key ?x file ?z)) ))
        (getr fieldnames)))))

(new-value/done
    (pop (list (getr value) (getr fieldnames)) t) )

;
; returns the fieldname and filename corresponding to the
found verb
;
(verb/
    (jump verb/end
        (or (string-equal (root-of (getr *)) "be")
            (string-equal (root-of (getr *)) "have")) )
    (jump verb/value
        (and (root-of (getr *))
            (eval (buildq (find field ?x trans +)
                (root-of (getr *))))) )

(verb/value
    (jump verb/end t
        (setr verb (root-of (getr *)))
        (setr fieldname
            (car (eval (buildq
                (find trans- (find field ?x trans +))
                verb))));
        (setr filename
            (car (eval (buildq
                (find file- (find member + file ?x))
                fieldname))))))

(verb/end
    (pop (list (getr fieldname) (getr filename)) t) )

;
; numeric adjective i.e. ... 20 year old ...;
(\(num\)-adj
    (push numeric t
        (setr fieldvalue *)
        (to \(num\)-adj/num) )
    (\(num\)-adj/num
        (to (\(num\)-adj/num/units)
            (eval (buildq (find field ?x unit +)
(root-of (getr *)) )
(setr filename
  (car (eval (buildq
    (find field- (find field ?x unit +))
    (root-of (getr *)) ))))) )

(num-adj/num/units
  (to (num-adj/end)
    (or (eval (buildq (find min-adj + field ?x) *))
      (eval (buildq (find max-adj + field ?z) *)) )
    (jump num-adj/end t) )

(num-adj/end
  (pop (list (getr filename) (getr fieldvalue)) t) )

;; is current item a key value
;;
(np/key
  (to (np/key/val)
    (eval (buildq (find field ?x value +) *))
    (setr filename
      (car (eval (buildq
        (find field- (find field ?x value +)) *))))
    (setr fieldvalue *) ) )

(np/key/val
  (jump np/key/end
    (eval (buildq (find key + file ?x) filename))) )

(np/key/end
  (pop (list (getr filename) (getr fieldvalue)) t) )

;; is current item the name of a boolean field
;
(boolean
  (to (boolean/end)
    (eval (buildq (find member + type boolean) *))
    (setr filename *)
    (setr fieldvalue (getr truthvalue)) )
  (to (boolean/synonym)
    (eval (buildq (find synonym + field ?x) *))
    (setr filename
      (car (eval
        (buildq
          (find field- (find synonym + field ?x)) *))))
    (setr fieldvalue (getr truthvalue)) ) )

(boolean/synonym
(jump boolean/end
  (buildq (find member + type boolean) filename) )
)

(boolean/end
  (pop (list (getr filename) (getr fieldvalue)) t) }

; is current item the name or synonym of a file
;
(np/filename
  (to (np/filename/end)
    (eval (buildq (find synonym + file ?x) *))
    (setr filename
      (car (eval
        (buildq
          (find file- (find synonym + file ?x)) *)) ))
    (push-filename (getr filename)) )
  (to (np/filename/end)
    (eval (buildq (find member + class filenames) *))
    (setr filename *)
    (push-filename (getr filename)) ) )
)

(np/filename/end
  (pop filename t) )

; is current item a filename or filename synonym
;
(fieldname
  (to (fieldname/end)
    (eval (buildq (find member + file ?x) *))
    (setr filename *)
  (to (fieldname/end)
    (eval (buildq (find field ?x synonym +) *))
    (setr filename
      (car (eval (buildq
        (find field-
          (find field ?x synonym +)) *)))) ) )
)

(fieldname/end
  (pop filename (not (eval (buildq (find key + file ?x) filename)))) )

; accept BE as an aux verb
;
(aux/
  (to (aux/end) (string-equal "be" (root-of (getr *)))) )

(aux/end
  (pop 'ok t) )

;
; parse a transitive verb phrase
;
(vp/)
  (to (vp/verb) (and (root-of (getr *)))
    (eval (buildq (find field ?x trans +) (root-of (getr *))) )))

  (setr fieldname
    (car (eval
      (buildq (find field- (find field ?x trans +)) (root-of (getr *)) ))))

  (setr filename
    (car (eval
      (buildq (find file- (find member + file ?x))
        fieldname)))) )

(vp/verb
  (cat det t (to vp/verb))
  (to (vp/verb/obj) (eval (buildq (find field + value +)
      fieldname *))
    (setr fieldvalue *)
  (push num-adj t
    (setr fieldvalue (cadr (getr *)))
    (to vp/verb/obj) )
  (cat prep t (to vp/verb))
  (to (vp/verb/obj)
    (not (eval (buildq (find field + value +)
      fieldname (getr *)))
    (setr fieldvalue *)
    (eval (buildq (assert field + value +)
      fieldname fieldvalue))
    (findordefine (getr fieldvalue)
      '(((ctgy . t))) ) ) )

(vp/verb/obj
  (jump vp/end t
    (setr update (append (getr update) (list
      (getr filename)
        '|
        (getr fieldname)
        '='
        (getr fieldvalue) )) ) )

(vp/end
  (pop (list (getr update)) t) )
; is current item a number
;
(numeric
  (to (numeric/end) (isnumber (getr *))
   (setr num *) ) )

(numeric/end
  (to (numeric/end) (eval (buildq (find field ?x unit +)
          (root-of (getr *)) )
   (pop num t) )

; handle parse errors
;
(parse-error
  (jump parse-error/action t
   (setr action
    (error-handler (getr error-condition)
      (getr *)) ) )

(parse-error/action
  (push abort-parse (eq (getr action) ’abort-parse)
   (setr response ’(**** parse aborted ****))
   (to parse-error/end) ) )

(parse-error/end
  (pop response t) )

; consume the remaining input
;
(abort-parse
  (to (abort-parse) (getr *))
  (pop ’t t) )

; checks to see if any predefined rules will be violated by
; update register
; needs sender’d retrieval, qualification, operation,
; update, join
;
(check-rules
  (jump check-rules/done (nullr update))
  (push check-rules/numeric
    (eval (buildq (find member + type numeric)
          (nthelem 3 (getr update)))
    (sendr update))
APPENDIX F: Post-Processor Source Listing

; PROGRAMMER: Michael Bessasparis
;
; DATE: December 9, 1990
;
; This program takes the information provided in the five
; registers: retrieval, qualification, operation, update, and
; join, and converts it to a dBase III program. It works for
; the operations add, change, delete, list, total, count, and
; yesno. It is also capable of retrieving data which is
; contained in two separate files. The information in the
; register join can specify more than two files, however
; because of the way dBase III programs work, I chose to
; simplify the programming by limiting it to two files. I
; wrote some of this program myself, however I did retain
; many functions written by Steve Safigan and Angel Diaz.
; Diaz wrote the post-processor for McMax which is very
; similar to dBase. So, with the exception of the update
; functions, this post-processor is nearly identical to the
; McMax one.

; top level of post processor.
;
; Programmer: Steve Safigan.
; Modified: to print statements which prepare dBase for a
; new program.
;
(defun post-process (retrieval qualification operation update join)
  (setq retrieval
        (convert-nodes-to-symbols retrieval))
(setq qualification
        (convert-nodes-to-symbols qualification))
(setq update (convert-nodes-to-symbols update))
(setq join (convert-nodes-to-symbols join))
(line-feed)
(line-feed)
(princ "Contents of IQL registers:")
(line-feed)
(princ " retrieval: ")
(princ retrieval)
(line-feed)
(princ " qualification: ")
(princ qualification)
(line-feed)
(princ " operation: ")
(princ operation)
(line-feed)
(princ " update: ")
(princ update)
(line-feed)
(princ " join: ")
(princ join)
(line-feed)
(line-feed)
(princ "clear all")
(line-feed)
(princ "set talk off")
(line-feed)
(princ "set heading off")
(line-feed)
(line-feed)
(setq line-number 0)
(use-statements
 (merge (get-filenames retrieval)
 (merge (get-filenames update)
 (merge (get-filenames qualification)
 (get-filenames join) ) ) ) )

(cond
 ((equal operation 'delete)
  (delete-statements retrieval
   qualification join))
 ((equal operation 'add)
  (add-statements qualification update join))
 ((equal operation 'change)
  (change-statements retrieval
   qualification update join))
 (t (retr-statements retrieval qualification
    operation join)))
)
(line-feed))

(defun convert-nodes-to-symbols (lst)
 (cond
  ((null lst) nil)
  ((typep (car lst) 'SNEPS:NODE)
   (cons (intern (princ-to-string (car lst)))
     (convert-nodes-to-symbols (cdr lst)) ) )
  (t (cons (car lst)
     (convert-nodes-to-symbols (cdr lst))))) )

(defun delete-statements (retrieval qualification join)
 (cond
  ((null join) ; only one file is being used
   (setq select (first-file qualification))
   (cond...))
; If any of the qualifications specify 'greatest or 'least,
; then get that information first.
(cond
  ((member 'greatest qualification)
   (find-greatest retrieval qualification select) )
  ((member 'least qualification)
   (find-least retrieval qualification select) )
  (t nil))
(line-feed)
(princ '|delete all |)
(princ ';|)
(line-feed)
(princ '|'    for |)
(qual-statements qualification select) )
(t ; CASE 2: Two files must be joined.
 ; If any of the qualifications specify 'greatest
 ; or 'least,
 ; then get that information first.
(cond
  ((member 'greatest qualification)
   (find-greatest retrieval qualification
   (eval (superlative-file qualification))))
  ((member 'least qualification)
   (find-least retrieval qualification
   (eval (superlative-file qualification)))) ) )
)

(defun d-statement (qual join)
  (setq root (car join))
  (setq leaf (nthelem 4 join))
  (locate-statement qual (eval leaf))
  (princ "join_var = ")
  (princ (join-field join (eval leaf)))
  (line-feed)
  (princ "select ")
  (princ (eval root))
  (line-feed)
  (princ "delete all ")
  (princ ";")
  (line-feed)
  (princ "    for ")
  (cond
   ((exist-qual-fields qual (eval root))
    (qual-statements qual (eval root))
    (princ " and ";")
    (line-feed)
    (princ "    ") ) ) )
(print-join-qualification join qual (eval root))
(line-feed) )

(defun change-statements (retrieval qualification
 update join)
 (cond
   ((null join) ; only one file is being used
    (setq select (first-file qualification))
    ; If any of the qualifications specify 'greatest or
    ; 'least,
    ; then get that information first.
    (cond
     ((member 'greatest qualification)
      (find-greatest retrieval
       qualification select) )
     ((member 'least qualification)
      (find-least retrieval
       qualification select) )
    (t nil) )
    (line-feed)
    (princ '|replace all |)
    (replace-statements update select)
    (princ '|;|)
    (line-feed)
    (line-feed)
    (princ '|    for |)
    (qual-statements qualification select) )
   (t ; CASE 2: Two files must be joined.
    ; If any of the qualifications specify 'greatest
    ; or 'least,
    ; then get that information first.
    (cond
     ((member 'greatest qualification)
      (find-greatest retrieval qualification
       (eval (superlative-file qualification)))))
     ((member 'least qualification)
      (find-least retrieval qualification
       (eval (superlative-file qualification)) ) )
    )
    (c-statement qualification update join) ) )

(defun c-statement (qual update join)
 (setq root (car join))
 (setq leaf (nthelem 4 join))
 (locate-statement (append qual ' (and) update)
   (eval leaf))
 (princ "join_var = ")
 (princ (join-field join (eval leaf)))
 (line-feed)
(line-feed)
(princ "select ")
(princ (eval root))
(line-feed)
(princ "replace all ")
(replace-statements update (eval root))
(cond
 ((exist-qual-fields update (eval leaf))
   (cond
    ((exist-qual-fields update (eval root))
     (princ " .and. ;")
     (line-feed)
     (princ 
       " ")
     (print-join-replace join update (eval root))
     )
     (princ ";")
     (line-feed)
     (princ " for ")
     (qual-statements qual (eval root))
     (cond
      ((exist-qual-fields qual (eval leaf))
       (cond
        ((exist-qual-fields qual (eval root))
         (princ " .and. ;")
         (line-feed)
         (princ " ")
         (print-join-qualification join
          update (eval root))
         )
         (line-feed)
       )
       ))
     )
     (defun add-statements (qual update join)
      (cond
       ((null join) ; only one file is being used
        (princ '|append blank|)
        (line-feed)
        (princ '|replace |)
        (replace-statements (append qual '(and) update)
         (eval (car qual)))
        (line-feed)
       )
       (t ; two files must be joined together
        (a-statement qual update join))
      )
    )
  )
  )
  )
(defun a-statement (qual update join)
  (setq root (car join))
  (setq leaf (nthelem 4 join))
  (locate-statement (append qual '(and) update)
   (eval leaf))
  (princ "join_var = ")
  (princ (join-field join (eval leaf)))
  (line-feed)
  (line-feed)
  (princ "select ")
  )
(princ (eval root))
(line-feed)
(princ "append blank ")
(line-feed)
(princ "replace ")
(replace-statements update (eval root))
(cond
  ((exist-qual-fields update (eval leaf))
   (cond
    ((exist-qual-fields update (eval root))
     (princ " .and. ;")
     (line-feed)
     (princ " ")
     (print-join-replace join update (eval root))
     (princ " .and. ;")
     (line-feed)
     (qual-statements qual (eval root))
     (cond
      ((exist-qual-fields qual (eval leaf))
       (cond
        ((exist-qual-fields qual (eval root))
         (princ " .and. ;")
         (line-feed)
         (princ " ")
         (print-join-qualification join update (eval root)))
       (line-feed))
      )))
   )))
)
(defun replace-statements (update selected-file)
  (cond
   ((eq selected-file (eval (car update)))
    (cond
     ; If the update is boolean then print in appropriate
     ; format.
     ((equal (nthelem 5 update) 'true)
      (princ (nthelem 3 update))
      (princ '| with .T.))
     ((equal (nthelem 5 update) 'false)
      (princ (nthelem 3 update))
      (princ '| with .F.))
     (t (princ (nthelem 3 update))
      (princ " ")
      (princ 'with)
      (princ " ")
      (princ " "))
    ; if its a numeric update then just print it.
    (cond
     ((isnumber (nthelem 5 update))
      (princ (nthelem 5 update)))
     ; Otherwise, its a character expression and requires
     (t ; quotation marks around the string.
      (quotation-marks))
    )))
141 (princ (nthelem 5 update))
    (quotation-marks) ) )
; If there are more updates for this file, then print
; "and."
    ; and continue through the list.
    (cond
        ((more-quals (nthcdr 6 update)
          selected-file)
         (princ " .and. ;") (line-feed)
         (princ " ")
         (replace-statements (nthcdr 6 update)
          selected-file))
        (t nil) )
    )
    (t (cond
        ((more-quals (nthcdr 6 update) selected-file)
         (replace-statements (nthcdr 6 update)
          selected-file))
        (t nil))))

; Generate USE statements.
;
; Programmer: Steve Saigan.
; Modified: to print a select statement before each "use".
;
(defun use-statements (filename-list)
    (cond
        ((null filename-list) nil)
        (t
            (setq line-number (1+ line-number))
            (set (car filename-list) line-number)
            (princ "select ")
            (princ line-number)
            (line-feed)
            (princ "use ")
            (princ (car filename-list))
            (line-feed)
            (line-feed)
            (use-statements (cdr filename-list))))

; generate statements which qualify the chosen files
; Take the first qualification of the list, if this
; statement qualifies the
; selected file, then print the qualification and continue
; on with the rest of
; the list. Otherwise, skip the qualification and proceed
; with the rest of the
; list. The function "is-number" is defined in the file
; Function.l.
;
; Programmer: Angel Diaz.
;
(defun qual-statements (qualification selected-file)
(cond
  ((null qualification) nil)
  ((either 'greatest 'least (list (car qualification)))
    (cond
      (equal selected-file
        (eval (cadr qualification)))
        (princ (nthelem 4 qualification))
        (cond
          ((member 'greatest qualification)
            (princ " >= m_greatest")
          )
          (t
            (princ " <= m_least")
          )
        )
        (cond
          ((more-quals (nthcdr 5 qualification) selected-file)
            (princ " .and. ;")
            (line-feed)
            (princ " ")
            (qual-statements
              (nthcdr 5 qualification)
              selected-file)
          )
          (t nil)))
    )
  )
)
(equal selected-file (first-file qualification))
  (cond
    ; If the qualification is boolean then print in appropriate format.
    ((equal (nthelem 5 qualification) 'true)
      (princ (nthelem 3 qualification))
    )
    ((equal (nthelem 5 qualification) 'false)
      (princ " .not. ")
      (princ (nthelem 3 qualification))
    )
    (t
      (princ (nthelem 3 qualification))
      (princ " ")
      (princ (nthelem 4 qualification))
      (princ " ")
    )
    ; if its a numeric qualification then just print it.
    ; Otherwise, its a character expression and requires (t ; quotation marks around the string.
    (isnumber (nthelem 5 qualification))
      (princ (nthelem 5 qualification)))
    )
    ; If there are more qualifications for this file,
    ; then print ".and."
143
; and continue through the list.
(cond
  ((more-quals (nthcdr 6 qualification)
    selected-file)
   (princ " .and. ;")
   (line-feed)
   (princ ")")
   (qual-statements (nthcdr 6 qualification)
    selected-file))
  (t nil)))
(t (qual-statements (nthcdr 6 qualification)
  selected-file)))

; Generate retrieval statements for all operations.
; If the join register is nil, then the programs are simple
; enough to generate
; within this function, otherwise, call the proper function.
;
; Programmer: Angel Diaz.
;
(defun retr-statements (retrieval qualification
operation join)
  (cond
   ((null retrieval) nil)
   ((null join); CASE 1: only one file is being used.
    ; If any of the qualifications specify 'greatest or
    ; 'least,
    ; then get that information first.
    (cond
      ((member 'greatest qualification)
       (find-greatest retrieval qualification
        (eval (car retrieval))))
      ((member 'least qualification)
       (find-least retrieval qualification
        (eval (car retrieval))))
      (t nil))
   (cond
    ((equal operation 'total)
     ; To total the data the operation 'sum
     ; must be used.
     (oper-statement retrieval
      qualification 'sum join
      (eval (car retrieval)))
     (line-feed)
     (princ "?m_sum")
    )
    ((equal operation 'yesno)
     ; Count the information,
     (oper-statement retrieval
     )
qualification 'count join
  (eval (car retrieval)))
(line-feed)
(princ "if m_count = 0")
(line-feed); then evaluate the
(princ "  text ")
(line-feed); results.
(princ "    no")
(princ " endtext")
(princ "else")
(princ "  text")
(princ "  yes")
(princ " endtext")
(princ "endif")

(t ; Then the operation is either list or count.
  (oper-statement retrieval qualification
    operation join
    (eval (car retrieval)))
  (cond
   ((equal operation 'count)
    ;If operation was a count,
   (line-feed) ; print the result.
   (princ "?m_count")
   (t nil))))))

(t ; CASE 2: Two files must be joined.

; If any of the qualifications specify 'greatest
; or 'least,
; then get that information first.
(cond
 ((member 'greatest qualification)
  (find-greatest retrieval qualification
   (eval (superlative-file qualification)))))
 ((member 'least qualification)
  (find-least retrieval qualification
   (eval (superlative-file qualification))))))

;These operations are more complex, so each one will be done
; using its own function.
(cond
 ((equal operation 'list)
  (list-statement retrieval
   qualification operation join))
 ((equal operation 'total)
  (total-statement retrieval
   qualification operation join))
 ((equal operation 'count)
  (count-statement retrieval
   qualification operation join))
(t ; operation equals yesno
  (yesno-statement retrieval
   qualification operation join)))))))

; Retrieve and display the information requested.
;
; Programmer: Angel Diaz.
;
(defun list-statement (retrieval qualification
  operation join)
  ; the loop to display the data should be done with the
  ; file that first
  ; occurs in the qualification register.
  (locate-statement qualification
   (first-file qualification))
  (princ "do while .not. eof()") (line-feed)
  (princ " join_var = ")
  (princ (join-field join (car qualification)))
  (line-feed)

  ; Make sure the information displayed within the loop
  ; occurs in the
  ; correct order.
  (cond
    ((equal (first-file qualification)
      (eval (car retrieval)))
     (oper-statement retrieval nil 'display
      nil (first-file qualification))
     (oper-statement retrieval qualification operation
      join
      (other-file join
       (first-file qualification)))
     (t
      (oper-statement retrieval qualification operation
       join
       (other-file join
        (first-file qualification)))
      (oper-statement retrieval nil 'display
       nil (first-file qualification)))
      (princ "select ")
      (princ (first-file qualification)) (line-feed)
      (princ "continue") (line-feed)
      (princ "enddo")
      (line-feed))

; Generates the retrieval program when two files are being
; joined,
; and adds together the data in the fields which meet the
; qualifications.
; Programmer: Angel Diaz.

(defun total-statement (retrieval qualification
operation join)
  (princ "m_total = 0") (line-feed)
  (locate-statement qualification
   (first-file qualification))
  (princ "do while .not. eof()") (line-feed)
  (princ " join_var = ")
  (princ (join-field join (car qualification)))
  (line-feed)
  (oper-statement retrieval qualification 'sum join
   (eval (car retrieval)))
  (princ " m_total = m_total + m_sum") (line-feed)
  (princ "select ")
  (princ (first-file qualification)) (line-feed)
  (princ "continue") (line-feed)
  (princ "endo")
  (line-feed)
  (princ ",?m_total")
  (line-feed))

; Generates the retrieval program when two files are being
; joined,
; and prints the total count meeting the qualifications.
;
; Programmer: Angel Diaz.

(defun count-statement (retrieval qualification
operation join)
  (princ "m_total = 0") (line-feed)
  (locate-statement qualification
   (first-file qualification))
  (princ "do while .not. eof()") (line-feed)
  (princ " join_var = ")
  (princ (join-field join (car qualification)))
  (line-feed)
  (oper-statement retrieval qualification operation join
   (eval (car retrieval)))
  (princ " m_total = m_total + m_count") (line-feed)
  (princ "select ")
  (princ (first-file qualification)) (line-feed)
  (princ "continue") (line-feed)
  (princ "endo")
  (line-feed)
  (princ ",?m_total")
  (line-feed))

; Generates the retrieval program when two files are being
; joined, and
; counts the occurrences that meet the specified conditions.
; it then determines
; if the information requested exists or not.
;
; Programmer: Angel Diaz.

(defun yesno-statement (retrieval qualification
operation join)
  (princ "m_total = 0") (line-feed)
  (locate-statement qualification
   (first-file qualification))
  (princ "do while .not. eof()") (line-feed)
  (princ "join_var = ")
  (princ (join-field join (car qualification))
   (line-feed)
  (oper-statement retrieval qualification 'count join
   (eval (car retrieval)))
  (princ " m_total = m_total + m_count") (line-feed)
  (princ "select ")
  (princ (first-file qualification)) (line-feed)
  (princ "continue") (line-feed)
  (princ "enddo")
  (line-feed)
  (princ "if m_total = 0") (line-feed)
  (princ " text ") (line-feed)
  (princ " no") (line-feed)
  (princ " endtext") (line-feed)
  (princ "else") (line-feed)
  (princ " text") (line-feed)
  (princ " yes") (line-feed)
  (princ " endtext") (line-feed)
  (princ "endif") (line-feed)
  (line-feed))

; Returns the field associated with a file in the join
; register for the selected
; file.
;
; Programmer: Angel Diaz.

(defun join-field (join select)
  (cond
   ((equal select (car join)) (caddr join))
   (t (nthelem 6 join)))))

; Generates the appropriate statements necessary to find the
; greatest value
; associated with the fieldname preceded by 'greatest,
; meeting the conditions
; specified in the qualification register.
(defun find-greatest (retrieval qualification select)
  (line-feed)
  (princ "m_greatest = 0")
  (line-feed)
  (locate-statement qualification select)
  (princ "do while .not. eof()")
  (line-feed)
  (princ "    if ")
  (princ (superlative-field qualification))
  (princ " > m_greatest")
  (line-feed)
  (princ "    m_greatest = ")
  (princ (superlative-field qualification))
  (line-feed)
  (princ "    endif")
  (line-feed)
  (princ "    continue")
  (line-feed)
  (princ "enddo")
  (line-feed))

; Generates the appropriate statements necessary to find the
; least value
; associated with the fieldname preceded by 'least, meeting
; the conditions
; specified in the qualification register.
;
; Programmer: Angel Diaz.
;
(defun find-least (retrieval qualification select)
  (line-feed)
  (princ "m_least = 9999999999999999")
  (line-feed)
  (locate-statement qualification select)
  (princ "do while .not. eof()")
  (line-feed)
  (princ "    if ")
  (princ (superlative-field qualification))
  (princ " < m_least")
  (line-feed)
  (princ "    m_least = ")
  (princ (superlative-field qualification))
  (line-feed)
  (princ "    endif")
  (line-feed)
  (princ "    continue")
  (line-feed)
  (princ "enddo")
  (line-feed))
; Returns the value associated with the first file in the register.
; Programmer: Angel Diaz.
(defun first-file (register)
  (cond
     ((null register) nil)
     ((equal (cadr register) '|.|
      (eval (car register)))
    (t (first-file (cdr register)))))))

; Returns the value of the other file in the join register from the one
; specified in the parameter select.
; Programmer: Angel Diaz.
(defun other-file (join select)
  (cond
     ((equal (first-file join) select)
      (eval (nthelem 4 join)))
    (t (eval (car join))))))

; Returns the first fieldname following the identifiers
; "greatest" or "least".
; Programmer: Angel Diaz.
(defun superlative-field (register)
  (cond
    ((null register)
      (princ "ERROR! superlative-field not found.
    ((equal 'greatest (car register)) (cadddr register))
    ((equal 'least (car register)) (cadddr register))
    (t (superlative-field (cdr register))))))

; Returns the file number corresponding to the superlative qualification.
; The list "register" must contain the keyword 'greatest or 'least.
; Programmer Angel Diaz
(defun superlative-file (register)
  (cond
    ((null register)
      (princ "ERROR! superlative-file not found."))
    ((equal 'greatest (car register)) (cadr register))
    ((equal 'least (car register)) (cadr register))
    (t (superlative-file (cdr register))))
)

; Check if any of the qualifications remaining are in the
; selected file.
;
; Programmer: Angel Diaz.
;
(defun more-quals (register selected-file)
  (cond
    ((null register) nil)
    ((equal 'greatest (car register))
      (more-quals (cdr register) selected-file))
    ((equal 'least (car register))
      (more-quals (cdr register) selected-file))
    ((equal '|.| (cadr register))
      (cond
        ((equal selected-file (eval (car register))) t)
        (t (more-quals (nthcdr 3 register)
                        selected-file)))
      (t (more-quals (cdr register) selected-file))))
)

; Return t if either x or y in l.
;
; Programmer: Angel Diaz.
;
(defun either (x y l)
  (cond
    ( (null l) nil)
    ( (member x l) t)
    ( (member y l) t)
    ( t nil )))

; Generates the operation statement passed in oper if there
; are any retrieval
; fields for the selected file.
;
; Programmer: Angel Diaz.
;
(defun oper-statement (retr quals oper join select)
  (cond
    ((null retr) nil)
    ((exist-retr-fields retr select)
; Check if data needs to be retrieved.
(line-feed)
(princ "select ")

; Select the appropriate file so the retrieved
(princ select)
; information will be obtainable.
(line-feed)
(princ oper)
; Print the operation passed to the function.
(princ ")
(cond
  ((eq oper 'count) t)
   (t (print-retrievals retr select)) )
(cond
  ((exist-qual-fields quals select)
   ; If the data is being qualified
   (princ ";")
   ; then print the conditions.
   (line-feed)
   (princ " for ")
   (qual-statements quals select)
   (cond
    ((null join) nil)
    ; If the qualification that would be given from the join
    ; register is already in the qualification statements
    ; then don't display it again.
    ((member (nthelem (times 3 select)
                   join) quals) nil)
    (t (princ ";") (line-feed)
      (princ ".and. ")
      (print-join-qualification join
       quals select))))
  ((not (null join)))
  ; After done with qualifications, check if
  (princ ";")
  ; need to qualify with a condition in another
  (line-feed) ; data file.
  (princ " for ")
  (print-join-qualification join
   quals select))
(t nil))
(cond
  ((equal oper 'count) (princ " to m_count"))
  ((equal oper 'sum) (princ " to m_sum"))
  (t nil))
(cond
  ((equal oper 'list) (princ ";")
   (line-feed)
   (princ "off ")
   (t nil) )
  (line-feed))
(t nil)))
; If the file is joined to another file, print the join
; field and set it
; equal to the "join variable" which is what is actually
; used to link the
; files.
;
; Programmer: Angel Diaz.
;
(defun print-join-qualification (join quals select)
  (cond
   ((null join) nil)
   (t (cond
       ((equal select (eval (car join)))
        (princ (nthelem 3 join))
        (princ " = join_var "))
       (t (princ (nthelem 6 join))
        (princ " = join_var "))))))

; Prints join variable for replace
;
(defun print-join-replace (join quals select)
  (cond
   ((null join) nil)
   (t (cond
       ((equal select (eval (car join)))
        (princ (nthelem 3 join))
        (princ " with join_var "))
       (t (princ (nthelem 6 join))
        (princ " with join_var "))))))

; Prints the fields for the selected file which need to be
; retrieved
; called by oper-statement.
;
; Programmer: Angel Diaz.
;
(defun print-retrievals (register selected)
  (cond
   ((null register) nil)
   ((equal selected (eval (car register)))
    (princ (caddr register))
    (cond
     ((exist-retr-fields
       (nthcdr 3 register) selected)
      (princ ", ")
      (print-retrievals (nthcdr 3 register) selected))
     (t (princ " ")
     (print-retrievals (nthcdr 3 register) selected)))))
; Return t if a field must be retrieved for the selected
; file.
;
; Programmer: Angel Diaz.
;
(defun exist-retr-fields (register select)
  (cond
   ((null register) nil)
   ((equal select (eval (car register))) t)
   (t (exist-retr-fields (nthcdr 3 register) select))))

; Generates a locate statement if there are qualifications
; for the
; selected file.
;
; Programmer: Angel Diaz.
;
(defun locate-statement (register select)
  (cond
   ((null register) nil)
   ((exist-qual-fields register select)
      (line-feed)
      (princ "select ")
      (princ select)
      (line-feed)
      (princ "locate for ")
      (qual-statements register select)
      (line-feed)
      (t nil)))

; Returns t if there are any qualifications for the selected
; file.
;
; Programmer: Angel Diaz.
;
(defun exist-qual-fields (register select)
  (cond
   ((null register) nil)
   ((equal 'greatest (car register))
      (cond
       ((equal select (eval (cadr register))) t)
       (t (exist-qual-fields (nthcdr 4 register) select))))
   ((equal 'least (car register))
      (cond
       ((equal select (eval (cadr register))) t)
       (t (exist-qual-fields (nthcdr 4 register) select))))
((equal select (eval (car register))) t)
(t (exist-qual-fields (nthcdr 6 register) select)))

; This function makes the code more readable by giving a
; name to character 10.
;
; Programmer: Angel Diaz.
;
(defun line-feed () (terpri))

; This function makes the code more readable by giving a
; name to character 34.
;
; Programmer: Angel Diaz.
;
(defun quotation-marks () (princ '"'))

(defun locate-value (field lst)
  (cond
       ((eq nil lst) nil)
       ((eq field (caddr lst)) (nthelem 5 lst))
       (t (locate-value field (cdr lst))))
    )

(defun remove-value (value lst)
  (cond
       ((atom lst) lst)
       ((eq value (caddr lst)) (nthcdr 5 lst))
       (t (append (myfirst 5 lst)
                  (remove-value value (nthcdr 5 lst))))
    )
)

(defun myfirst (num lst)
  (cond
       ((eq num 0) nil)
       (t (cons (car lst) (myfirst (sub1 num) (cdr lst)))))))
APPENDIX G: Response Expert Source Listing

; decide on kind of response to generate
; (respond

(push delete-response (eq (getr operation) 'delete)
  (sendr retrieval)
  (sendr qualification)
  (sendr update)
  (sendr join)
  (setr string *)
  (to gen/done) )

(push change-response (eq (getr operation) 'change)
  (sendr retrieval)
  (sendr qualification)
  (sendr update)
  (sendr join)
  (setr string *)
  (to gen/done) )

(push add-response (eq (getr operation) 'add)
  (sendr retrieval)
  (sendr qualification)
  (sendr update)
  (sendr join)
  (setr string *)
  (to gen/done) )

(push count-response (eq (getr operation) 'count)
  (sendr retrieval)
  (sendr qualification)
  (sendr join)
  (setr string *)
  (to gen/done) )

(push yesno-response (eq (getr operation) 'yesno)
  (sendr retrieval)
  (sendr qualification)
  (sendr join)
  (setr string *)
  (to gen/done) )

(push total-response (eq (getr operation) 'total)
(sendr retrieval)
(sendr qualification)
(sendr join)
(setr string *)
(to gen/done )

(push list-response (eq (getr operation) 'list)
  (sendr retrieval)
  (sendr qualification)
  (sendr join)
  (setr string *)
  (to gen/done )
)

(jump gen/done t
  (setr string '(error)) ) )

;; stop generation if no more to output
;;
(gen/done
  (pop string t) )

(delete-response
  (push d-r t
    (sendr string)
    (sendr qualification)
    (sendr retrieval)
    (sendr update)
    (sendr join)
    (setr string *)
    (to delete-response/done ) ) )

(delete-response/done
  (pop string t) )

(d-r
  (jump d-r/ihd t
    (addr string 'I 'have 'deleted) ) )

(d-r/ihd
  (jump d-r/done
    (eval (buildq (find key + file ?x)
      (nthelem 3 (getr qualification)) )))
    (addr string (nthelem 5 (getr qualification))))
  (push get-subj t
    (sendr qualification)
    (sendr retrieval)
    (sendr update)
    (sendr join)
    (addr string *)
  )
(to d-r/subj) )

(d-r/subj
 (jump d-r/subj/that (not (nullr qualification))
   (addr string 'that))
 (jump d-r/done (nullr qualification))))

(d-r/subj/that
 (push r-vp t
   (sendr string)
   (sendr mods (getr qualification))
   (sendr num 'plur)
   (setr string *)
   (to d-r/done) ) )

(d-r/done
 (pop string t))

(change-response
 (push a-r
   (eval (buildq (find key + file ?x)
     (nthelem 3 (getr qualification)))
   (sendr string)
   (sendr qualification)
   (sendr retrieval)
   (sendr update)
   (sendr join)
   (setr string *)
   (to change-response/done) )
 (push c-r
   (not (eval
     (buildq (find key + file ?x)
       (nthelem 3 (getr qualification)))
   (sendr string)
   (sendr qualification)
   (sendr retrieval)
   (sendr update)
   (sendr join)
   (setr string *)
   (to change-response/done) ) )

(change-response/done
 (pop string t))

(c-r
 (jump c-r/iut t
   (addr string 'I 'understand 'that) ) )

(c-r/iut
(push get-subj t
  (sendr qualification)
  (sendr retrieval)
  (sendr update)
  (sendr join)
  (addr string *)
  (to c-r/subj) ) )

(c-r/subj
  (jump c-r/subj/that (not (nullr qualification))
   (addr string 'that) )
  (jump c-r/vp (nullr qualification)) )

(c-r/subj/that
  (push r-vp t
   (sendr string)
   (sendr mods (getr qualification))
   (sendr num 'plur)
   (setr string *)
   (to c-r/vp) ) )

(c-r/vp
  (push r-vp t
   (sendr string)
   (sendr mods (getr update))
   (sendr num 'plur)
   (setr string *)
   (to c-r/vp/vp) ) )

(c-r/vp/vp
  (pop string t) )

(add-response
  (push a-r t
   (sendr string)
   (sendr qualification)
   (sendr retrieval)
   (sendr update)
   (sendr join)
   (setr string *)
   (to add-response/done) ) )

(add-response/done
  (pop string t) )

(a-r
  (jump a-r/iut t
   (addr string 'I 'understand 'that) ) )

(a-r/iut
(jump a-r/key t
   (addr string (nthelem 5 (getr qualification))) ) )

(a-r/key
   (push r-vp t
      (sendr string)
      (sendr mods (getr update))
      (sendr num 'sing)
      (setr string *)
      (to a-r/done) ) )

(a-r/done
   (pop string t) )

(list-response
   (push list-answer t
      (sendr string)
      (sendr qualification)
      (sendr retrieval)
      (sendr join)
      (setr string *)
      (to list-response/done) ) )

(list-response/done
   (pop string t) )

(list-answer
   (jump la/det
      (and (getr qualification)
         (eval
            (buildq (find key + file ?x)
               (nthelem 3 (getr qualification)) ) ) )
      (setr subj-num 'sing)
      (addr string 'The) )
   (jump la/det t
      (setr subj-num 'plur)
      (addr string 'The) ) )

(la/det
   (push get-retr t
      (sendr subj-num)
      (sendr retrieval)
      (sendr string)
      (setr string *)
      (to la/det/retr) ) )

(la/det/retr
   (jump la/det/retr/of t
      (addr string 'of) ) )
(la/det/retr/of
  (jump la/r-vp
   (and (getr qualification)
    (eval
     (buildq (find key + file ?x)
      (nthelem 3 (getr qualification)) )) )
   (addr string (nthelem 5 (getr qualification)))
  (push get-subj t
   (sendr qualification)
   (sendr retrieval)
   (sendr update)
   (sendr join)
   (addr string (getr *))
  (to la/det/retr/of/subj)) )

(la/det/retr/of/subj
 (jump la/subj/that (not (nullr qualification))
  (addr string 'that))
 (jump la/r-vp (nullr qualification)))

(la/subj/that
 (push r-vp t
  (sendr string) (sendr mods (getr qualification))
  (sendr num 'plur)
  (setr string '*)
  (to la/r-vp)) )

(la/r-vp
 (jump la/r-vp/follow
  (and (eq 3 (length (getr retrieval)))
   (eq 'sing (getr subj-num))
  (addr string 'is))
 (jump la/r-vp/follow t
  (addr string 'are)))

(la/r-vp/follow
 (jump la/done t
  (addr string '<answer>))

(la/done
 (pop string t))

(get-retr
 (jump get-retr/field (eq (getr subj-num) 'sing)
  (addr string (nthelem 3 (getr retrieval)))
  (setr retrieval (nthcdr 3 (getr retrieval)))
 (jump get-retr/field (eq (getr subj-num) 'plur)
  (addr string
   (wordize 'plur (nthelem 3 (getr retrieval))))
  (setr retrieval (nthcdr 3 (getr retrieval))))
)
(get-retr/field
    (jump get-retr (not (nullr retrieval))
        (addr string 'and) )
    (jump get-retr/done t) )

(get-retr/done
    (pop string t) )

(total-response
    (push ta t
        (sendr string)
        (sendr qualification)
        (sendr retrieval)
        (sendr join)
        (setr string *)
        (to total-answer/done ) )
)

(total-answer/done
    (pop string t) )

(ta
    (jump ta/det t
        (setr num 'plur)
        (addr string 'The) )
)

(ta/det
    (push get-retr t
        (sendr subj-num 'sing)
        (sendr string)
        (sendr retrieval)
        (setr string *)
        (to ta/det/field) )
)

(ta/det/field
    (jump ta/det/field/of t
        (addr string 'of) )
)

(ta/det/field/of
    (jump ta/vp
        (eval
            (buildq (find key + file ?x)
                (nthelem 3 (getr qualification)) ))
            (addr string (nthelem 5 (getr qualification))) )
        (push get-subj t
            (sendr qualification)
            (sendr retrieval)
            (sendr update)
            (sendr join)
            (addr string *)
            (to ta/det/field/of/subj) )
    )
)
(ta/det/field/of/subj
  (jump ta/det/field/of/subj/that
    (not (nullr qualification))
    (addr string 'that) )
  (jump ta/vp (nullr qualification)) )

(ta/det/field/of/subj/that
  (push r-vp t
    (sendr string)
    (sendr mods (getr qualification))
    (sendr num)
    (setr string *)
    (to ta/vp) ) )

(ta/vp
  (jump ta/done t
    (addr string 'is '<answer>)) )

(ta/done
  (pop string t) )

(yesno-response
  (push yn-answer t
    (sendr string)
    (sendr qualification)
    (sendr retrieval)
    (sendr join)
    (setr string *)
    (to yn-answer/done) ) )

(yn-answer/done
  (pop string t) )

(yn-answer
  (jump yn-answer/yn t
    (addr string 'Yes 'there 'are) )
  (jump yn-answer/yn t
    (addr string 'No 'there 'are 'not 'any)) )

(yn-answer/yn
  (push get-subj t
    (sendr qualification)
    (sendr retrieval)
    (sendr update)
    (sendr join)
    (addr string *)
    (to yn-answer/subj) ) )

(yn-answer/subj
  (jump yn-vp (not (nullr qualification)))
(addr string 'that) )
(jump yn-vp/done (nullr qualification))  )

(yn-vp
  (push r-vp t
    (sendr string)
    (sendr mods (getr qualification))
    (sendr num 'plur)
    (setr string *)
    (to yn-vp/done) )
)

(yn-vp/done
  (pop string t) )

(count-response
  (push cr-np t
    (sendr string)
    (sendr retrieval)
    (sendr join)
    (setr string *)
    (to count-response-np/done) )
)

(cr-np
  (push get-subj t
    (sendr qualification)
    (sendr retrieval)
    (sendr update)
    (sendr join)
    (addr string '<answer> (getr *))
    (setr num 'plur)
    (liftr num)
    (to cr-np/done) )
)

(cr-np/done
  (pop string t) )

(count-response-np/done
  (push r-vp t
    (sendr string)
    (sendr mods (getr qualification))
    (sendr num)
    (setr string *)
    (to count-response-np/vp/done) )
)

(count-response-np/vp/done
  (pop string t) )

(r-vp
  (push r-pred (not (nullr mods)))
(sendr string) (sendr mods) (sendr num)
(setr string *)
(to r-vp/end) )
(jump r-vp/end (nullr mods)) )

(r-vp/end
 (jump r-vp (member (car (getr mods)) '(and))
 (addr string (car (getr mods)))
 (setr mods (cdr (getr mods))) )
 (pop string (nullr mods)) )

(r-pred
 (push r-verb t
 (sendr string) (sendr mods) (sendr num)
 (setr string *)
 (liftr mods)
 (to r-pred/verb) ) )

(r-pred/verb
 (push r-obj t
 (sendr string)
 (sendr mods)
 (setr string *)
 (liftr mods)
 (to r-pred/verb/end) ) )

(r-pred/verb/end
 (pop string t) )

(r-verb
 (jump r-verb/end
 (and (eq (getr num) 'plur)
 (eval
 (buildq (find member + type boolean)
 (nthelem 3 (getr mods)) )))
 (addr string 'are) )

(jump r-verb/end
 (and (eq (getr num) 'sing)
 (eval (buildq (find member + type boolean)
 (nthelem 3 (getr mods)) )))
 (addr string 'is) )

(jump r-verb/end
 (and (eq (getr num) 'plur)
 (eval (buildq (find member + type alphanumeric)
 (nthelem 3 (getr mods)) )))
 (addr string 'have) )

(jump r-verb/end
165
(and (eq (getr num) 'sing)
   (eval (buildq (find member + type alphanumeric)
                 (nthelem 3 (getr mods))) ))
(addr string 'has) )

(jump r-verb/end
 (and (eq (getr num) 'plur)
   (eval (buildq (find member + type numeric)
                 (nthelem 3 (getr mods))) ))
(addr string 'have) )

(jump r-verb/end
 (and (eq (getr num) 'sing)
   (eval (buildq (find member + type numeric)
                 (nthelem 4 (getr mods))) ))
(addr string 'has) )

(jump r-verb/end
 (and (eq (getr num) 'plur)
   (eval (buildq (find member + type numeric)
                 (nthelem 4 (getr mods))) ))
(addr string 'have) )

(r-verb/end
 (pop string t) )

(r-obj
 (push r-boolean
  (eval (buildq (find member + type boolean)
                (nthelem 3 (getr mods))) ))
       (sendr string)
       (sendr mods)
       (setr string *)
       (setr mods (nthcdr 5 (getr mods)))
       (liftr mods)
       (to r-obj/end) )
(push r-alpha
  (eval (buildq (find member + type alphanumeric)
                (nthelem 3 (getr mods))) ))
       (sendr string)
       (sendr mods)
       (setr string *)
(setq mods (nthcdr 5 (getr mods)))
(liftf mods)
(to r-obj/end)
(push r-numeric
 (eval (buildq (find member + type numeric)
 (nthelem 3 (getr mods)) ))
 (sendr string)
 (sendr mods)
 (setq string *)
 (setq mods (nthcdr 5 (getr mods)))
 (liftf mods)
 (to r-obj/end)
 (push r-est (or (equal (car (getr mods)) 'greatest)
 (equal (car (getr mods)) 'least) ))
 (sendr string)
 (sendr mods)
 (setq string *)
 (setq mods (nthcdr 4 (getr mods)))
 (liftf mods)
 (to r-obj/end))
(r-obj/end
 (pop string t))

(r-boolean
 (push get-synonym (eq (nthelem 5 (getr mods)) 'false)
 (sendr field (nthelem 3 (getr mods)))
 (setq synonym *)
 (addr string 'not (getr synonym))
 (to r-boolean/end)
 (push get-synonym (eq (nthelem 5 (getr mods)) 'true)
 (sendr field (nthelem 3 (getr mods)))
 (setq synonym *)
 (addr string (getr synonym))
 (to r-boolean/end))
(r-boolean/end
 (pop string t))

(r-alpha
 (push get-synonym t
 (sendr field (nthelem 3 (getr mods)))
 (setq synonym *)
 (addr string (get-det (getr synonym)) (getr synonym))
 (to r-pp))

(r-pp
 (jump r-pp/prep t
 (addr string 'of)))
(r-pp/prep
  (jump r-pp/end t
    (addr string (nthelem 5 (getr mods))) ) )

(r-pp/end
  (pop string t) )

(r-est
  (jump r-est/the t
    (addr string 'the) ) )

(r-est/the
  (jump r-est/the/wrd t
    (addr string (car (getr mods))) ) )

(r-est/the/wrd
  (push get-synonym t
    (sendr field (nthelem 4 (getr mods)))
    (setr synonym *)
    (addr string (getr synonym))
    (to r-est/end) ) )

(r-est/end
  (pop string t) )

(r-numeric
  (push get-synonym t
    (sendr field (nthelem 3 (getr mods)))
    (setr field (nthelem 3 (getr mods)))
    (setr synonym *)
    (addr string (get-det (getr synonym)) (getr synonym))
    (to r-numeric/field) ) )

(r-numeric/field
  (jump r-numeric/comp (eq (nthelem 4 (getr mods)) '=")
    (addr string 'of) )
  (jump r-numeric/comp (eq (nthelem 4 (getr mods)) '">)
    (addr string 'greater 'than) )
  (jump r-numeric/comp (eq (nthelem 4 (getr mods)) '<")
    (addr string 'less 'than) )
  (jump r-numeric/comp (eq (nthelem 4 (getr mods)) '">=
    (addr string 'of 'at 'least) )
  (jump r-numeric/comp (eq (nthelem 4 (getr mods)) '<="
    (addr string 'of 'at 'most) ) )

(r-numeric/comp
  (push get-unit t
    (sendr field)
    (setr unit *)
)
(adûr string (nthelem 5 (getr mods)) (wordize 'plur
(getr unit)))
(to r-numeric/end ) )

(r-numeric/end
(pop string t) )

(get-subj
(jump get-subj/end
(or (eq (car (getr qualification)) 'GREATEST)
   (eq (car (getr qualification)) 'LEAST) )
   (setr subj (cadr (getr qualification))) )
(jump get-subj/end (nullr join)
   (setr subj (or (car (getr retrieval))
                  (car (getr qualification)) )))
(jump get-subj/end (not (nullr join))
   (setr subj (car (getr join))) ) )

(get-subj/end
(pop subj t) )

(get-unit
(jump get-unit/end t
   (setr unit
      (car (eval (buildq
                      (find unit- (find field + unit ?x))
                      field)))) ) )

(get-unit/end
(pop unit t) )

(get-synonym
(jump get-synonym/end t
   (setr synonym
      (cond
       ((eval (buildq (find field + synonym ?x) field))
        (car (eval (buildq
                      (find field- (find field + synonym ?x))
                      field)))) )
       (t (getr field))) ) ) )

(get-synonym/end
(pop (getr synonym) t) )
APPENDIX H: Sample System Output

This example demonstrates the operation of the interface. The interview is from Safigan's work [Safigan 1987]. Note, the query parser is currently being ported to the new version of SNePS. Therefore, no responses to queries are demonstrated here.

The database used for this demonstration consists of two relations. The first relation consists of patient information, including diagnosis. The second relation describes cares associated with certain conditions. The relations are joined through the diagnosis/condition field.

The PATIENTS relation consists of the following nine fields:

1. NAME - Alphanumeric key field.
2. SEX - Alphanumeric field. Has values of MALE and FEMALE.
3. COMPLAINT - Alphanumeric field. Has corresponding verb COMPLAIN.
4. AMBULATORY - Boolean field.
5. DIAGNOSIS - Alphanumeric field. Provides the link to the CARES relation.
6. DOCTOR - Alphanumeric field. Has corresponding verb SEE.
7. RACE - Alphanumeric field. Has predefined value WHITE.
8. SKIN - Alphanumeric field.
9. AGE - Numeric field. Has maximizing adjective OLD, minimizing adjective YOUNG, units of measure YEARS, lower bound of 0, and upper bound of 200.
The CARES relation consists of three fields:

1. CONDITION - Alphanumeric, key field.
2. DIET - Alphanumeric field.
3. ACTIVITYLEVEL - Alphanumeric field.
Welcome to SNePS-2.1

File START.L is now the source of input.

CPU time : 0.03   GC time : 0.00

* (^ (lexin 'data.lex))

undef ined- (NIL NIL NIL NIL)
THEIR THEY SHE HE IT HER HER HER HAD HAS HAVE A AN THE OF ABOUT AT FOR IN TO WITH BY BE IS ARE AM WAS BEEN DO DOES DID DONE NUMBER AMOUNT QUANTITY WHAT WHERE WHEN WHO WHOSE WHICH THAT NO NOT)

CPU time : 0.87   GC time : 0.00

* (^ (load 'function.l))

; Loading /home/bessaspa/FUNCTION.L. (T)

CPU time : 3.25   GC time : 0.00

* (^ (load 'dbpost.l))

; Loading /home/bessaspa/DBPOST.L. (T)

CPU time : 5.25   GC time : 0.00

* (^ (atnin 'thesis.l))
Enter new database filename or end: PATIENT
Done! Enter synonym for PATIENT or end: PATIENTS
Done! Enter synonym for PATIENT or end: END
Enter corresponding verb for PATIENT or end: END
Enter field name or end: NAME
Done! Enter synonym for NAME or end: END
Is this field a key field? YES
Done! Enter corresponding verb for NAME or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: SEX
Done! Enter synonym for SEX or end: END
Is this field a key field? NO
Enter corresponding verb for SEX or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: MALE
Done! Enter data value for field or end: FEMALE
Done! Enter data value for field or end: END
Enter field name or end: COMPLAINT
Done! Enter synonym for COMPLAINT or end: END
Is this field a key field? NO
Enter corresponding verb for COMPLAINT or end: COMPLAIN
Done! Enter corresponding verb for COMPLAINT or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: AMBULATORY
Done! Enter synonym for AMBULATORY or end: END
Is this field a key field? NO
Enter corresponding verb for AMBULATORY or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? B
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter field name or end: DIAGNOSIS
Done! Enter synonym for DIAGNOSIS or end: END
Is this field a key field? NO
Enter corresponding verb for DIAGNOSIS or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: DOCTOR
Done! Enter synonym for DOCTOR or end: END
Is this field a key field? NO
Enter corresponding verb for DOCTOR or end: SEE
Enter corresponding verb for RACE or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: WHITE
Done! Enter data value for field or end: END
Enter field name or end: SKIN
Done! Enter synonym for SKIN or end: END
Is this field a key field? NO
Enter corresponding verb for SKIN or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: AGE
Done! Enter synonym for AGE or end: END
Is this field a key field? NO
Enter corresponding verb for AGE or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? N
Done! Enter the lower bound or end: 0
Done! Enter the upper bound or end: 200
Done! Enter the name a proper unit of measure or end: YEAR
Done! Enter the name a proper unit of measure or end: YEARS
Done! Enter the name a proper unit of measure or end: END
Enter the name of a maximizing adjective or end: OLD
Done! Enter the name of a maximizing adjective or end: END
Enter the name of a minimizing adjective or end: YOUNG
Done! Enter the name of a minimizing adjective or end: END
Enter field name or end: END
Enter new database filename or end: CARE
Done! Enter synonym for CARE or end: END
Enter corresponding verb for CARE or end: END
Enter field name or end: CONDITION
Done! Enter synonym for CONDITION or end: END
Is this field a key field? YES
Done! Enter corresponding verb for CONDITION or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: DIET
Done! Enter synonym for DIET or end: END
Is this field a key field? NO
Enter corresponding verb for DIET or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: ACTIVITYLEVEL
Done! Enter synonym for ACTIVITYLEVEL or end: END
Is this field a key field? NO
Enter corresponding verb for ACTIVITYLEVEL or end: END
Is field (A)lpha, (N)umeric, or (B)oolean? A
Done! Does this field modify the data base file? (Y/N) Y
Done! Enter data value for field or end: END
Enter field name or end: END
Enter new database filename or end: END
Enter origin file of link or end: PATIENTS
Enter origin field of link: DIAGNOSIS
Enter destination file of link: CARE
Enter destination field of link: CONDITION
Done! Enter origin file of link or end: END
(T)

CPU time : 23.20     GC time : 0.00

* (^{parse})

ATN parser initialization...

Trace level = 0.

Beginning at state 'S'.

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

: ~^~

Enter Lisp Read/Eval/Print loop. Type ~^~ to continue
-->(setq *parse-trees* t)

T
-->(^)

: MARY IS A WHITE FEMALE PATIENT WHO WAS COMPLAINING OF CHESTPAIN.

By MARY are you referring to
<1> NAME
<2> OTHER

Contents of IQL registers:
retrieval: NIL
qualification: (PATIENTS . NAME = MARY)
operation: ADD
update: (PATIENTS . SEX = FEMALE AND
         PATIENTS . RACE = WHITE AND
         PATIENTS . COMPLAINT = CHESTPAIN)
join: NIL
clear all
set talk off
set heading off
select 1
use PATIENTS
append blank
replace NAME with "MARY" .and. ;
     SEX with "FEMALE" .and. ;
     RACE with "WHITE" .and. ;
     COMPLAINT with "CHESTPAIN"

Time (sec.): 12.717
Resulting parse:
(I UNDERSTAND THAT "MARY" HAS A SEX OF "FEMALE" AND HAS A
RACE OF "WHITE" AND HAS A COMPLAINT OF "CHESTPAIN")
SHE IS 65 YEARS OLD.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: CHANGE
  update: (PATIENTS . AGE = 65)
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all AGE with 65;
  for NAME = "MARY"

Time (sec.): 4.083
Resulting parse:
(I UNDERSTAND THAT "MARY" HAS A AGE OF "65" "YEARS")
: SHE SEES DRJONES.

Contents of IQL registers:
   retrieval: NIL
   qualification: (PATIENTS . NAME = MARY)
   operation: CHANGE
   update: (PATIENTS . DOCTOR = DRJONES)
   join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all DOCTOR with "DRJONES";
   for NAME = "MARY"

   Time (sec.): 5.334
   Resulting parse:
   (I UNDERSTAND THAT "MARY" HAS A DOCTOR OF "DRJONES")
: SHE IS NOT AMBULATORY.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: CHANGE
  update: (PATIENTS . AMBULATORY = FALSE)
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all AMBULATORY with .F.;
  for NAME = "MARY"

Time (sec.): 2.917
Resulting parse:
(I UNDERSTAND THAT "MARY" IS NOT "AMBULATORY")
SHE HAS A DIAGNOSIS OF ACUTEMI.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: CHANGE
  update: (PATIENTS . DIAGNOSIS = ACUTEMI)
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all DIAGNOSIS with "ACUTEMI";

    for NAME = "MARY"

Time (sec.): 3.8
Resulting parse:
(I UNDERSTAND THAT "MARY" HAS A "DIAGNOSIS" OF "ACUTEMI")
: SHE HAS WHITE SKIN.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: CHANGE
  update: (PATIENTS . SKINCOLOR = WHITE)
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all SKINCOLOR with "WHITE";
  for NAME = "MARY"

  Time (sec.): 4.883

Resulting parse:
(I UNDERSTAND THAT "MARY" HAS A SKINCOLOR OF "WHITE")
: TOM IS A WHITE PATIENT.

By TOM are you referring to
<1> NAME
<2> CONDITION
<3> OTHER

By WHITE are you referring to
<1> SKINCOLOR
<2> RACE
<3> OTHER

Contents of IQL registers:
    retrieval: NIL
    qualification: (PATIENTS . NAME = TOM)
    operation: ADD
    update: (PATIENTS . RACE = WHITE)
    join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

append blank
replace NAME with "TOM" .and. ;
    RACE with "WHITE"

Time (sec.): 8.6
Resulting parse:
(I UNDERSTAND THAT "TOM" HAS A RACE OF "WHITE")
: **ACUTEMI HAS A DIET OF LOWCAL.**

By ACUTEMI are you referring to
<1> CONDITION
<2> OTHER

Contents of IQL registers:
  retrieval: NIL
  qualification: (CARE . CONDITION = ACUTEMI)
  operation: ADD
  update: (CARE . DIET = LOWCAL)
  join: NIL

clear all
set talk off
set heading off

select 1
use CARE

append blank
replace CONDITION with "ACUTEMI" .and. ;
  DIET with "LOWCAL"

Time (sec.): 6.666
Resulting parse:
(I UNDERSTAND THAT "ACUTEMI" HAS A "DIET" OF "LOWCAL")
: DIABETES HAS A DIET OF ADA.

By DIABETES are you referring to
<1> CONDITION
<2> OTHER

Contents of IQL registers:
  retrieval: NIL
  qualification: (CARE . CONDITION = DIABETES)
  operation: ADD
  update: (CARE . DIET = ADA)
  join: NIL

clear all
set talk off
set heading off

select 1
use CARE

append blank
replace CONDITION with "DIABETES" .and. :
    DIET with "ADA"

Time (sec.): 6.417
Resulting parse:
(I UNDERSTAND THAT "DIABETES" HAS A "DIET" OF "ADA")
: DIABETES ACTIVITYLEVEL IS ADLIB.

Contents of IQL registers:
  retrieval: NIL
  qualification: (CARE . CONDITION = DIABETES)
  operation: CHANGE
  update: (CARE . ACTIVITYLEVEL = ADLIB)
  join: NIL

clear all
set talk off
set heading off

select 1
use CARE

replace all ACTIVITYLEVEL with "ADLIB";
  for CONDITION = "DIABETES"

  Time (sec.): 3.967
Resulting parse:
  (I UNDERSTAND THAT "DIABETES" HAS A "ACTIVITYLEVEL" OF "ADLIB")
ACUTEMI has an ACTIVITYLEVEL of BEDREST.

Contents of IQL registers:
- retrieval: NIL
- qualification: (CARE . CONDITION = ACUTEMI)
- operation: CHANGE
- update: (CARE . ACTIVITYLEVEL = BEDREST)
- join: NIL

clear all
set talk off
set heading off

select 1
use CARE

replace all ACTIVITYLEVEL with "BEDREST";

    for CONDITION = "ACUTEMI"

Time (sec.): 4.05
Resulting parse:
(I UNDERSTAND THAT "ACUTEMI" HAS A "ACTIVITYLEVEL" OF "BEDREST")
: MARY HAS A DIET OF ADA.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: CHANGE
  update: (CARE . DIET = ADA)
  join: (PATIENTS . DIAGNOSIS CARE . CONDITION)

clear all
set talk off
set heading off

select 1
use PATIENTS

select 2
use CARE

select 2
locate for DIET = "ADA"
join_var = CONDITION

select 1
replace all DIAGNOSIS with join_var ;
  for NAME = "MARY"

Time (sec.): 4.167
Resulting parse:
(I UNDERSTAND THAT "MARY" HAS A "DIET" OF "ADA")
: ADLIB PATIENTS ARE AMBULATORY.

Contents of IQL registers:
  retrieval: NIL
  qualification: (CARE . ACTIVITYLEVEL = ADLIB)
  operation: CHANGE
  update: (PATIENTS . AMBULATORY = TRUE)
  join: (PATIENTS . DIAGNOSIS CARE . CONDITION)

clear all
set talk off
set heading off

select 1
use PATIENTS

select 2
use CARE

select 2
locate for ACTIVITYLEVEL = "ADLIB"
join_var = CONDITION

select 1
replace all AMBULATORY with .T. ;
  for DIAGNOSIS = join_var

Time (sec.): 5.467
Resulting parse:
(I UNDERSTAND THAT PATIENTS THAT HAVE A ACTIVITYLEVEL OF "ADLIB" ARE "AMBULATORY")
: ADA PATIENTS SEE DRWHITE.

Contents of IQL registers:
  retrieval: NIL
  qualification: (CARE . DIET = ADA)
  operation: CHANGE
  update: (PATIENTS . DOCTOR = DRWHITE)
  join: (PATIENTS . DIAGNOSIS CARE . CONDITION)

clear all
set talk off
set heading off

select 1
use PATIENTS

select 2
use CARE

select 2
locate for DIET = "ADA"
join_var = CONDITION

select 1
replace all DOCTOR with "DRWHITE" ;
  for DIAGNOSIS = join_var

  Time (sec.): 6.167
Resulting parse:
(I UNDERSTAND THAT PATIENTS THAT HAVE A DIET OF "ADA" HAVE A
DOCTOR OF "DRWHITE")
JEFF HAS A DIET OF ADA.

By JEFF are you referring to
<1> CONDITION
<2> OTHER

By JEFF are you referring to
<1> NAME
<2> OTHER

Contents of IQL registers:
retrieval: NIL
qualification: (PATIENTS . NAME = JEFF)
operation: ADD
update: (CARE . DIET = ADA)
join: (PATIENTS . DIAGNOSIS CARE . CONDITION)

clear all
set talk off
set heading off

select 1
use PATIENTS

select 2
use CARE

select 2
locate for DIET = "ADA"
join_var = CONDITION

select 1
append blank
replace DIAGNOSIS with join_var .and. ;
NAME = "JEFF"

Time (sec.): 6.55
Resulting parse:
(I UNDERSTAND THAT "JEFF" HAS A "DIET" OF "ADA")
: DELETE MARY.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MARY)
  operation: DELETE
  update: NIL
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

delete all;
  for NAME = "MARY"

  Time (sec.): 1.85
Resulting parse:
  (I HAVE DELETED "MARY")
: DELETE FEMALE PATIENTS.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . SEX = FEMALE)
  operation: DELETE
  update: NIL
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

delete all ;
  for SEX = "FEMALE"

  Time (sec.): 3.984
Resulting parse:
(I HAVE DELETED PATIENTS THAT HAVE A SEX OF "FEMALE")
: DELETE PATIENTS WHO SEE DRWHITE.

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . DOCTOR = DRWHITE)
  operation: DELETE
  update: NIL
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

delete all ;
  for DOCTOR = "DRWHITE"

  Time (sec.): 3.883
Resulting parse:
  (I HAVE DELETED PATIENTS THAT HAVE A DOCTOR OF "DRWHITE")
: DELETE THE OLDEST PATIENT.

Contents of IQL registers:
  retrieval: NIL
  qualification: (GREATEST PATIENTS . AGE)
  operation: DELETE
  update: NIL
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

m_greatest = 0

select 1
locate for AGE >= m_greatest
do while .not. eof()
  if AGE > m_greatest
    m_greatest = AGE
  endif
  continue
endo

delete all ;
  for AGE >= m_greatest

  Time (sec.): 3.483
Resulting parse:
(I HAVE DELETED PATIENTS THAT HAVE THE GREATEST AGE)
: DELETE THE OLDEST ADA PATIENT.

Contents of IQL registers:
    retrieval: NIL
    qualification: (CARE . DIET = ADA AND GREATEST PATIENTS . AGE)
    operation: DELETE
    update: NIL
    join: (PATIENTS . DIAGNOSIS CARE . CONDITION)

clear all
set talk off
set heading off

select 1
use PATIENTS

select 2
use CARE

m_greatest = 0

select 1
locate for AGE >= m_greatest
do while .not. eof()
    if AGE > m_greatest
        m_greatest = AGE
    endif
    continue
endo

select 2
locate for DIET = "ADA"
join_var = CONDITION
select 1
delete all ;
    for AGE >= m_greatest and ;
        DIAGNOSIS = join_var

Time (sec.): 5.35
Resulting parse:
(I HAVE DELETED PATIENTS THAT HAVE A DIET OF "ADA" AND HAVE THE GREATEST AGE)
MIKE IS 300 YEARS OLD.

I'm sorry, 300 is an unacceptable value for AGE. The values for AGE must be between 0 and 200. Please enter the correct value for AGE now: 30

Contents of IQL registers:
  retrieval: NIL
  qualification: (PATIENTS . NAME = MIKE)
  operation: CHANGE
  update: (PATIENTS . AGE = 30)
  join: NIL

clear all
set talk off
set heading off

select 1
use PATIENTS

replace all AGE with 30;

  for NAME = "MIKE"

  Time (sec.): 4.933
Resulting parse:
(I UNDERSTAND THAT "MIKE" HAS A AGE OF 30 "YEARS")

^end

ATN Parser exits...

CPU time : 126.77  GC time : 0.00

* (lisp)

"End of SNePS"
<cl> (exit)
; Exiting Lisp
VITA

Michael J. Bessasparis was born May 1, 1965 in Elizabeth, New Jersey. He received a B.S. in Computer Science from the University of Pittsburgh in 1987. He received an M.S. in Computer Science from Virginia Polytechnic Institute and State University in 1990. He is now attending the United States Army Signal Officer Advanced Course at Fort Gordon, Georgia.