THE PLANES SYSTEM: NATURAL LANGUAGE ACCESS TO A LARGE DATA BASE

Annual Progress Report

by

David L. Waltz, Timothy Finin, Fred Green,
Forrest Conrad, Bradley Goodman, and George Hadden

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

The work described in this report was supported by the Office of Naval Research under Contract Number N00014-67-A-0305-0026.

TABLE OF CONTENTS

1. PLANES - A Natural Language Front End For a Large Data Base

A prime obstacle for non-technical people who wish to use computers has been the need to either learn a special language for communicating with the machine or communicate via an intermediary. We feel that the time is ripe for computers to be equipped for natural language systems which can be used by persons who are not trained in any special computer language. In order for such systems to be of value to a casual user, the systems must tolerate simple errors, must embody a degree of "common sense," must have a relatively large and complete vocabulary for the subject matter to be treated, must accept a wide range of grammatical constructions, and of course must be capable of providing the information and computations requested by the user.

We are developing such a system called PLANES (for Programmed LANguage-Based Enquiry System) at the University of Illinois Coordinated Science Laboratory [Gabriel and Waltz (1974), Waltz (1975a), Waltz (1976)]. PLANES includes an English language front end with the ability to understand and explicitly answer user requests and to carry on clarifying dialogues with him, as well as the ability to answer vague or poorly defined questions. We are also building a library of associated programs which includes functions for recognizing patterns within the data base and for alerting a user when certain patterns of data occur which are of interest to him. This work is being carried out using a subset of the U.S. Navy 3-M data base of aircraft maintenance and flight data, although the ideas can be directly applied to other record-based data bases, both military and non-military.

1.1 The Goal of PLANES

Our main goal is to allow a non-programmer to obtain information from a large data base with no prior training or experience. A system to realize this goal (1) must be able to understand to a substantial degree a user's natural language and (2) must be able to help guide and educate the user to formulate requests in a form that the system can understand (see Codd [1974]).

1.1.1 Subgoals of PLANES

We have formulated a number of subgoals which we feel are important for realizing our main goal:

(1) The system must accept a user's "natural" English input, possibly including complex syntactic constructions, abbreviations, pronoun reference, and ellipsis (i.e. omission of one or more words that can be obviously understood in context).

(2) The system must provide explicit answers to questions, and not merely retrieve a file which somewhere contains the answer. The system should phrase its answer in a clear manner, including units or dimensions of numerical answers and a description of the answer values. Whenever possible, graphical output is desirable.

(3) The system must be tolerant of minor errors, e.g. spelling and grammatical errors; it should suggest corrections for user approval whenever possible, and should in general be able to continue processing of the corrected request without requiring a complete retyping of the request.

(4) The system should use clarifying dialogues for several purposes:

(a) to feed back its understanding of the user's request, so that the user can feel confident that the system has understood his request.

(b) to ask the user pointed questions about portions of a request which it does not understand, in the hope of evoking a paraphrase it can understand.

(c) to add new words, phrases and sentences to the system's knowledge base.

(d) to provide appropriate HELP file information in the event of user errors or direct requests for help.

(e) to provide information about the system's capabilities, abbreviations it knows, general contents of the data base, and other such information to help orient a new user.

(5) Such a system should be convenient to use:

(a) it should be interactive and on-line,

(b) it should operate rapidly. One minute seems to us to be a critical length of time: if one in general gets answers in less than a minute, one retains a level of involvement and interest in the interactive process; past a minute, interest begins to wane, and boredom or impatience sets in.

(c) the system should require a minimum of typing; abbreviations and ellipsis should be routinely handled.

(6) The system should be relatively easy to extend, both within its own world, and to new data bases and domains of discourse.

It is unlikely that any design can simultaneously satisfy all these subgoals in an optimum manner. The first version of PLANES, described in the bulk of this report, was not particularly easy to extend, either within our world or to a new world. The second version, on which we are currently working, is vastly easier to extend. We have not yet achieved completely satisfying solutions either for the subgoal of providing good dialogue capability, nor for the subgoal or being able to gracefully continue from a partially understood sentence. Overall, our system will at this point (May 1976) answer probably no more than 25% of a novice user's questions. We expect this figure to improve dramatically over the next few months.

1.1.2 Examples of PLANES' Operation

All examples in this section are taken from actual transcripts of PLANES' operation. Figure 1.1 is a complete transcript of the operation of the current planes system on a single question. Note that the paraphrase and answer are expressed in a form which is not as intelligible as an English form would be. We intend to soon augment the system to provide English paraphrase and answer forms.

Figure 1.2 shows an extended session, with all but the question and answer suppressed. Figure 1.2 illustrates the ability of the system to handle ellipsis (i.e. information omitted which is to be understood in context).

```
Please enter your question.....
     HOW MANY FLIGHTS HOURS DID PLANE 0003 LOG IN JAN 73

parsing......
 {cpu time was 1.74 seconds, real time was 5.95 seconds.
I have understood your request as:
     (COUNT TIME
            (NET NIL)
            (FLY (PLANE (PRONOUN NIL)
                        (TYPE PLANE)
                        (BUSER 3.)
                        (PLNEG NIL)
                        (PLDAM NIL)
                        (PLMAI NIL))
                 NIL)
            (TIME (DATE (MONTH (1. 0. 0.))
                        (DAY NIL) (YEAR 73.)) NIL))

Interpreting.....
 {cpu time was 0.29 seconds, real time was 3.71 seconds.

I have interpreted your request as follows:
     (FIND  ALL
            ((V 0))
            ((SUM (V TOTHRS)))
            (AND (EQU (V ACTDATE) 301.)  (EQU  (V BUSER) 3.))
            NIL)

Evaluating.....
 {cpu time was 6.29 seconds, real time was 16.9 seconds.

        ((SUM TOTHRS) = 33.)
```

Figure 1.1 Complete transcript of PLANES' answer to a user query

```
)) How many flights did the A7 with tail number 003 make in
   January, 1973?
((SUM TOTFLTS) = 17.)

)) How many flights did plane 003 make in Feb. 73?
((SUM TOTFLTS) = 1.)

)) During April?
((SUM TOTFLTS) = 8.)

)) March?
((SUM TOTFLTS) = 13.)

)) All of 1973?
((SUM TOTFLTS) = 39.)
```

Figure 1.2 Partial transcript of dialogue showing PLANES' handling of ellipsis

Figure 1.3 shows PLANES' ability to provide answers in a graphical form. NORHRS stands for "Not Operationally Ready HouRS"; ACTDATE stands for "ACTion DATE"; 2001 is the first day of the year 1972.

Figure 1.4 shows a short dialogue. PLANES first asks for time period information necessary for performing its data base search. It then detects a misspelling and suggests two plausible corrections to the user. Once the user selects the appropriate correction by typing "2", the system is able to continue processing without requiring further typing.

Figure 1.5 shows PLANES' ability to answer general questions about its contents.

Further examples of PLANES' performance are given in the following sections, and an example of what we expect PLANES to do eventually is given in Appendix B.

1.2 The PLANES World

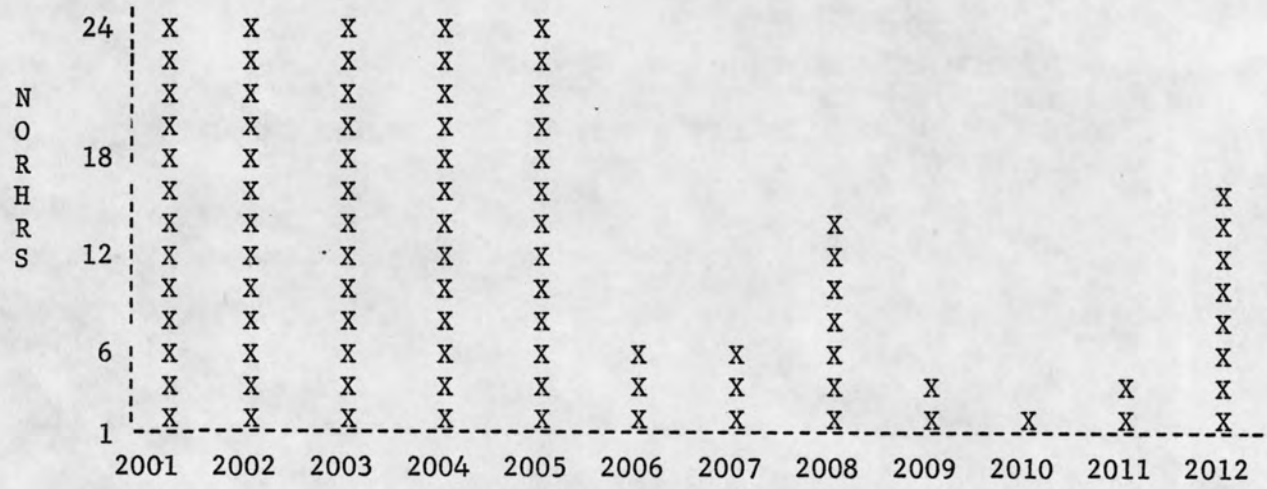
In this section I describe the general environment in which the language understanding programs operate. This environment includes the data base, the user, and the user's range of queries.

1.2.1 The PLANES Data Base

We have obtained a data base from the Navy 3-M Data Base for Aircraft, Mechanicsburg, PA., consisting of complete records of aircraft maintenance and flight information for 48 A7 and F4 aircraft, extending over a period of two years. Each time a plane is serviced, a record is made including such information as the time and duration of the maintenance, who performed it, what action was taken, which parts were used, the manufacturers of these parts, whether or not the service was scheduled or unscheduled, and so on. Records on the number of flights and the number of hours in the air are also kept for each plane. There are roughly forty different record formats which occur in the data base, each containing between ten and twenty separate fields, where each field encodes information like the date of the action, type of aircraft, serial number of the aircraft, type of malfunction, component serviced, the work station performing maintenance and so on. Our data base is described in detail in Appendices D, E, and F.

)) Plot how much maintenance occurred to plane 2 between Jan. 1 and Jan. 12, '72.

```
Evaluating.....
 {cpu time was 29.32 seconds, real time was 38.11 seconds.
```



ACTDATE

MIN = 1.0 AVERAGE = 14.0 MAX = 24.0

Figure 1.3 PLANES' graphical presentation of an answer

```
>> Which Skyhawks logged less than 10 hours
   PLEASE ENTER A SPECIFIC TIME PERIOD:

>> between Jan. 1 and Jule 1 '73.
   I DON'T KNOW THE MEANING OF JULE
   PERHAPS IT'S MISSPELLED?
   PLEASE ENTER A NUMBER CORRESPONDING TO ONE OF THE FOLLOWING:
   1. JUNE
   2. JULY
   3. NONE OF THE ABOVE

>> 2.
   OK

<Answers question>
```

Figure 1.4 Example of PLANES' dialogue and spelling correction abilities

```
Please enter your question.....
  ⟩⟩  What types of aircraft are there.

parsing......


THE FOLLOWING PLANES WILL BE RECOGNIZED BY THE SEMANTIC NET:

        (1) A3
        (2) A7
        (3) F4
        (4) F11
        (5) CHEROKEE
        (6) PHANTOM
        (7) SKYHAWK

    The above planes may be further specified by giving the
    "tail number" (i.e. BUSER, BUNO, Bureau Serial Number, etc.)
    along with the name of the series.

{cpu time was 2.59 seconds, real time was 5.75 seconds.
```

Figure 1.5 PLANES' ability to answer questions about its contents.

The 48 aircraft in our data base are divided into three groups:

(1) 24 planes which crashed or sustained major damage in accidents involving mechanical failures;

(2) 12 planes with bad maintenance records; and

(3) 12 planes with good maintenance records.

"Good" and "bad" records were judged by comparing the ratio of the number of NOR (Not Operationally Ready) hours to the number of flight hours. A high ratio represents a bad record while a low ratio corresponds to a good record.

In addition, we have summaries of maintenance and flight data for all F4 and A7 aircraft for the same two year period, so that we can have some basis for classifying events as "normal" or "unusual."

The PLANES data base contains on the order of 10^8 bits, and occupies about one third of a DEC RP04 disc pack in compressed form. This data base, while quite large, represents only a fraction of the entire 3-M data base, which now contains on the order of 10^{11} bits (10 years' complete data on all U.S. Navy aircraft, plus summaries).

1.2.2 Helpful Factors in the PLANES World

A number of factors contribute to making our problem much easier to solve than the general problem of understanding unconstrained natural language.

(1) Lack of ambiguity - Relatively few words and virtually no sentences in the PLANES world are ambiguous. The only ambiguous words we have been able to find are "wing" (meaning "a squadron" or "part of a plane") and "flight" (meaning "a flying event" or an adjective, as in "flight computer" or "flight director"). This means that if PLANES can find any interpretation at all for a request, it is in all likelihood to correct interpretation.

(2) Small vocabulary - Our current system has about 1100 words. We estimate that 2000 words will cover 90% or more of all requests made by users with at least a little prior experience with PLANES.

(3) Only two modes - PLANES is always either answering a question from the data base or attempting to help a user express his request in a form PLANES can understand. In general the system need not deal with declarative sentences.

(4) People do not type complex sentences - The increasing likelihood of making typing errors in lengthy requests, the increasing likelihood that long requests will baffle a program in some aspect, and general laziness all contribute to keeping input requests short and simple in construction. Malhotra (1975) performed an experiment in which non-programmers thought that they were communicating with an intelligent program, when in fact they were interacting with another person who would respond appropriately to any input. He found that 10 simple sentence types covered 78% of all input requests, and that another 10 would handle all but 10% of the requests.

(5) Less than 100% answer rate is acceptable - We feel that a 90% answer rate without rephrasing would be adequate to keep a user's interest and provide a practical and useful system. It is possible that even a lower rate might be acceptable.

(6) We have a good idea of what potential users would like to know - The Navy has a made a study [NALDA/NAILSC (1974)] of all the requests made to 3-M data base during a one month period, and the frequency with which various requests were made. We thus had a good idea about where to concentrate our initial efforts, and the order in which to proceed. A summary of this study is given in Appendix F.

1.2.3 Not-So-Helpful Factors in The PLANES World

(1) The System must contain a great deal of specialized knowledge - one of our major realizations has been that a small number of general rules cannot suffice to "translate" natural English requests into data base queries. Consider the sentences:

(S1) "Which A7 has the worst maintenance record?

(S2) "Find any common factors of plane numbers 37 and 78."

Clearly the system must contain special programs to compile a "maintenance record", and special knowledge to judge its "goodness"; the system must know that having the same digit as the fourth element of their serial numbers does not constitute a "common factor", but that similar event sequences are important.

(2) Each request may be expressed in a great many different ways - Clearly, if users are encouraged to sit at a console with little or no prior training or instruction, and if the system is expected to understand enough of a user's input to keep his interest and perform useful actions from the beginning, then the system must be able to make some sense of a large number of types of queries, and a wide range of syntactic constructions.

1.3 Operation of PLANES

The processing of a user's request is divided into four main phases: parsing, interpretation, evaluation, and response (see Figure 1.6).

(1) The first phase, the parsing, is accomplished by matching the input against request patterns stored in a large ATN network [Woods (1970)]. This network defines a "semantic grammar" [Brown and Burton (1975)] which maps the users request into an internal representation, the paraphrase language. The grammar is "semantic" in that it encorporates semantic and pragmatic knowledge as well as syntactic knowledge.

(2) In the interpretation phase, the paraphrase language representation of the user's request is translated into a 'program' to generate the data to answer the request. This stage contains specific knowledge about the data bases. It knows, for example, which sub-parts of the data base contain certain relations and the particular codes used to represent certain facts. The program is constructed out of primitives of the query language.

(3) The evaluation phase uses the program generated in the previous stage to search the data base and construct an answer. This portion is

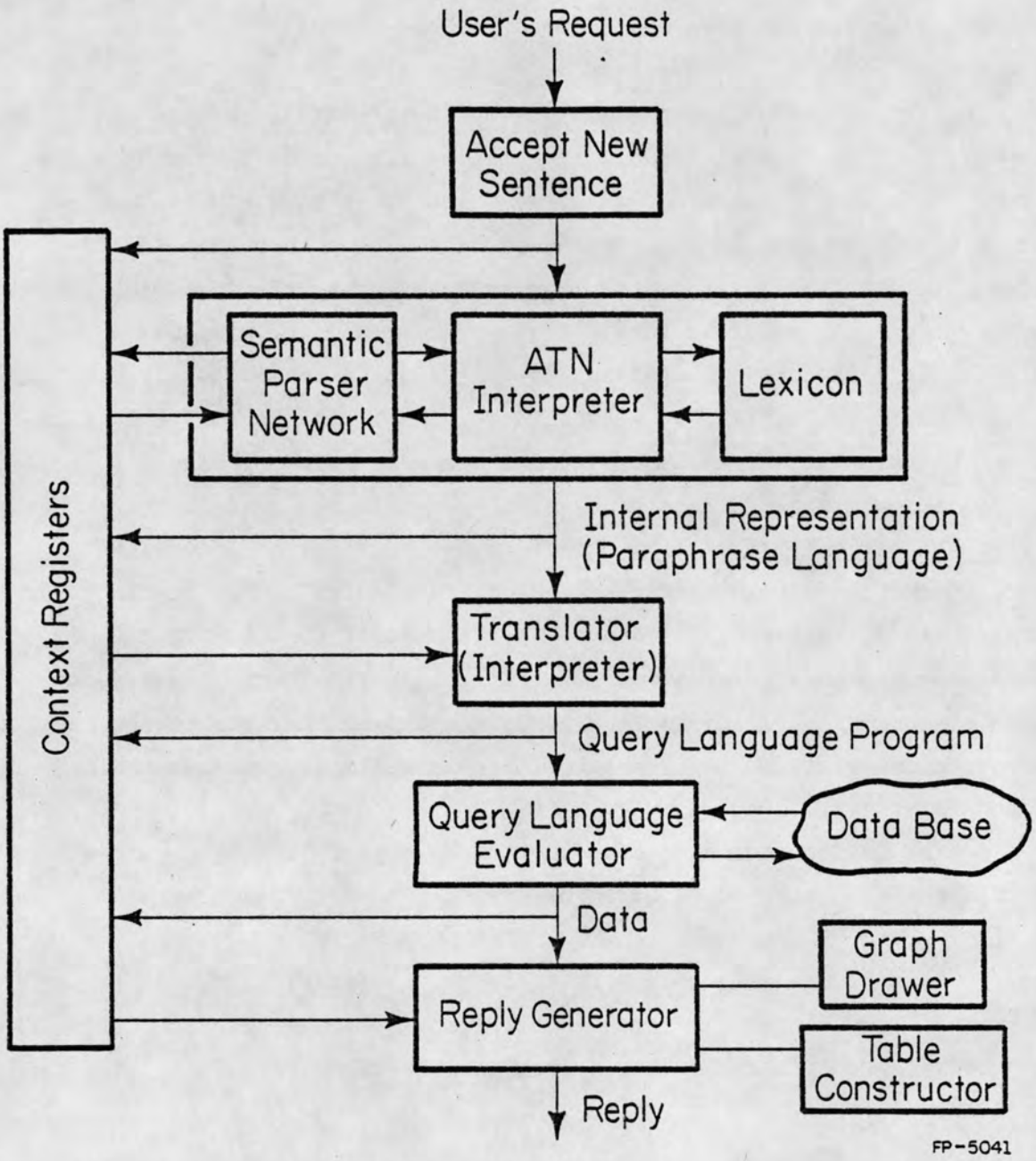


Figure 1.6 An overview of the PLANES system

based on a relational data model [Codd (1970), Date (1975)], and is implemented along lines described by Palermo (1972).

(4) The evaluation portion passes the resulting data to the response generator. This module can display the answer in one of three forms: as a simple number or list, as a graph, or as a table. The choice of output form can be determined by the user through a direct request (e.g. "Draw a graph of...) or by a set of heuristics which attempt to find the most "natural" form. A graph, for example, will be generated only if the data consists of a set of tuples which can be interpreted as a function of two variables. Furthermore, the number of tuples must lie within certain bounds, so that the entire result will fit on a CRT screen.

At each stage of the process, the results are sent to the history keeper which manages a set of stacks of relevant information. These stacks contain the results of each stage (e.g. user's request, paraphrase, etc.), syntactic components (e.g. subject, object, etc.), and semantic/contextual information (e.g. time specifications, plane specifications, etc.). This information is made available for resolving anaphoric reference, supplying phrases deleted through ellipsis, and generating responses.

The entire process can be aborted by any of the major stages. If this is done, a suitable error message is generated and the user is asked to reform either part or all of his request.

Each of the main phases is described in detail in the following sections. For greater clarity, we will trace a request through all phases of processing. Assume that the system has already successfully answered the question:

(Q1.3.0) "Which A7s logged less than 5 flight hours in February 1973?"

We then give PLANES the following request:

(Q1.3.1) "Which ones logged betwen 10 and 20 flight hours in Jan?".

1.3.1 Parsing

In the PLANES system, "parsing" involves three operations:

(1) putting all words and phrases in canonical form and correcting spelling;

(2) matching against prestored request patterns and setting context registers (history keepers); and
(3) constructing a paraphrase of the input request. These operations are explained in more detail in the following sections.

1.3.2 Putting Words and Phases in Canonical Form and Correcting Spelling

The parser first checks to make sure that each word of the input is known by the system. Roots and inflection markers are substituted for inflected words, canonical words are substituted for synonyms, and single words are substituted for certain phrases (i.e. "USA" for "the United States of America"). If a given input word cannot be found in the dictionary, then the spelling correction module is called. This module attempts to find dictionary entries "close" to the input word (see section 4.2 for a description of the spelling correction module); if one or more candidates are found, the user is given an appropriate message, and if one of the system's guesses is correct, it is inserted in place of the misspelled word. If no candidate words are found, or if none of the candidates is the properly spelled word, the system calls a word adding module to try to add the user's word to the dictionary.

Example 1.3.2 Given our question:

(Q1.3.1) "Which ones logged betwen 10 and 20 flight hours in Jan?"

the parser first notes that "betwen" is not in the dictionary; the spelling corrector finds that "between" is the most similar dictionary entry, and types back:

(A1.3.1) Is betwen a misspelling of between? Type y or n.

Once the user types "y", the system substitutes "between" for "betwen", and proceeds. "(Log past)" is substituted for "logged",

"flighthours" is substituted for "flight hours", and "January" is substituted for "Jan." Thus the output of this first operation is:

(Q1.3.2) Which ones (log past) between 10 and 20 flighthours in January?

1.3.3 Matching Against Prestored Request Patterns and Setting Context Registers

This phase comprises the heart of the language understanding process. It is here that pronoun reference and ellipsis are resolved, and here too that much of the overall programming effort for the system has been expended.

These are three large program portions of interest here:

(1) the prestored request network;

(2) the subnets; and

(3) the context registers.

The prestored request network is made up of pattern-action pairs, some simplified examples of which are shown in Figure 1.7. If the input matches a pattern, then the action corresponding to that pattern is executed. The action can be a data base search, the selection of a HELP file, or any other program we wish to associate with an input pattern. Many different patterns can point to a single action (as is the case with A1 in figure 1.7.).

Each of the starred pattern elements in Figure 1.7 (e.g. *planetype) must match a noun, verb or prepositional phrase which has a specific meaning (e.g. a type of aircraft, a period of time, etc.). Pattern elements preceded by a number sign (#log) match any synonymous word or phrase (e.g. logged, had, recorded, etc.).

The matching of starred elements of the patterns with phrases in the input is handled by subnets. Each subnet is a phrase parser which matches only phrases with a specific meaning. Some examples of phrases which the subnet for *planetype would match are: "A7", "phantom", "phantom or skyhawk", "ones" (where some type of aircraft is an appropriate referent of "ones") and "A7s which crashed in May".

Pattern	Matching Sentence	Action
(P1) Which *planetype #log *flighthours	(S1) Which A7s logged between 10 and 20 flight hours in Jan 1973?	(A1) Generate paraphrase from context register values
(P2) Which *planetype #log *flights	(S2) Please tell me which planes had 20 or more flights?	(A1)
(P3) How many *flighthours #log by *planetype	(S3) Find the number of flight hours logged by plane 76 during Feb. 1974.	(A1)
(P4) Which kinds of *data-items #exist	(S4) Which kinds of planes do you know about?	(A2) List *data-items from HELP files.

Figure 1.7 Sample pattern-action pairs

Some of the sentences (besides Q1.3.1) which match pattern P1 of Figure 1.7 are:

(Q1.3.3) Find which phantoms had fewer than 15 flight hours in Feb. 1974.

(Q1.3.4) Please tell me which skyhawks and F4s logged no flight hours.

(Q1.3.5) Which A7s that crashed in May had 50 or more flight hours in April?

Whenever a subnet matches a phrase, it sets the values of its corresponding context register, which acts as a history keeper. Context registers are used for pronoun reference and ellipsis; if some item(s) in a request have been left unspecified or replaced by pronouns, the context register values from the previous request are used to supply the missing information or the referent of a pronoun (see example 1.3.3). There are also context registers for the last request, last paraphrase, last query language form, and last answer. Context registers are implemented as stacks which are pushed down with each new request. While we have not yet done so, PLANES could conceivably retrieve an earlier context by saying something like:

(Q1.3.6) Earlier we were talking about skyhawks.

Time is handled in a special manner. Since time phrases can occur in any question, time phrases are not considered part of the patterns (see Fig. 1.7) but are searched for at the beginning or end of a request, or after the verb. There are special subnets and context registers for time phrases.

Quantifiers (e.g. "first", "rest", "more than", "largest", etc.) are handled by a special subnet as are qualifiers (e.g. the underlined words in "A7s which crashed in May").

"Noise words" are matched and essentially discarded. By "noise words", we mean phrases like "please tell me", "can you tell me", "would you let me know", "could you find", etc. The next section includes a description of the processing of such phrases.

A paraphrase of the first pattern found which matches the input is fed back to the user for his approval, and if the match is correct, the action associated with this pattern is executed. If the first match is incorrect, PLANES currently fails; eventually a more sophisticated presentation to the user of alternative interpretations will be implemented.

Example 1.3.3

Let us now follow our example through; the input to the matching portion from the previous portion is:

(Q1.3.2) Which ones (log past) between 10 and 20 flighthours in January?

This request matches pattern P1 in Figure 1.7, and causes context registers to be set as follows:

register	value
*oper	FINDALL (due to which)
*planetype	(pronoun ones)
*obj	flighthours
*quant-obj	(>10, <20)
*time-period	January

There is an important thing to note here: the system has solved the pronoun reference problem for the phrase "which ones" by finding the phrase type (i.e. *planetype) which makes sense in the context of the given sentence. There is only one pattern which matches "which ?X log flighthours": the pattern "which *planetype #log *flighthours." This allows us to conclude that ?X refers to *planetype. (While pilots could log flight hours, there is no pilot data in our world.) Each pattern can thus be viewed as representing a simple meaningful "concept" which the system knows. Thus whenever constituents of a sentence are missing

(as in ellipsis) or replaced by pronouns or referential phrases, the system is able to suggest what type of phrase is necessary to complete the concept by finding all the patterns which match the rest of the sentence. If only one matches, we are done; if more than one matches, then reference to which constituents were present in the previous sentence is usually adequate to decide; otherwise, the user can be given a set of possibilities from which to choose the appropriate referents for each phrase. The system can conclude that sentences like "How many malfunctions logged more than 10 flight hours." are meaningless because all phrases are recognized but no matching pattern exists.

Any system for answering questions which is to distinguish meaningful from meaningless requests must store information about which concepts are possible, whether by patterns (as in PLANES or in PARRY [Colby et al. (1974)]), by verb meaning structures as in case grammars [Fillmore (1968), Celce-Murcia (1972), Bruce (1975)], by conceptual dependency diagrams [Schank (1973)] or by preference semantics systems [Wilks (1975)].

Continuing with the example, once pattern P1 has been matched, the action A1 associated with P1 is retrieved. A1 in this case merely says to construct a paraphrase expression from the context register values, using the translator. The result is shown in figure 1.8.

```
(FINDALL (PLANE (PRONOUN T)
                (TYPE A7)
                (BUSER NIL))
         (NEG NIL)
         (FLY HOURS ((<20.)(>10.)))
         (TIME (DATE (MONTH (1. 0. 0.))
                     (DAY NIL)
                     (YEAR 73.))))
```

Figure 1.8 Paraphrase language representation of Q1.3.1.

This expression can be read as follows:

(1) FINDALL is a search function which means "find all items in the data base satisfying the specifications following." The context register *oper is set to FINDALL when the word "which" is matched in the original question.

(2) The three lines beginning with "PLANE ..." specify the item to be searched for and returned as a value, in this case A7s. (PRONOUN T) means that pronoun (or phrase) reference occurred in the request. Note that the value "A7" has been inserted for plane type. This value was obtained from the context register values from the previous sentence, and thus completes the handling of pronoun reference for the phrase "which ones". (BUSER NIL) means that no specific serial numbers of planes appeared in the request.

(3) (NEG NIL) means that the sentence is not negated.

(4) The three lines beginning "(TIME ...") specify that the entire first month of 1973 is to be considered. Here again the value of the year (1973) was not specified in the request, and so had to be obtained from the *time-period context register value of the previous request. This is another example of the handling of ellipsis (or automatic filling in of information understood in the context of the question) where the relevant context here is the preceding dialogue. It should be clear that similar mechanisms are used to handle both ellipsis and pronoun reference problems.

In our current system, the entire "paraphrase" is fed back to the user. In the near future, we intend to feed back an English version of this information, e.g. "Find all planes which had less than 20 and more than 10 flight hours during January 1973."

1.3.4 Implementation of Prestored Request Patterns and Subnets

Noun phrase parsing is based on Winograd's (1972) analysis of noun phrases. Both prestored request networks and subnets are stored as ATNs (Augmented Transition Networks) [Woods (1970)]. See Appendix A for a summary description of the use and implementation of ATNs.

Instead of storing and matching against a list of patterns (like the arrangement shown in Figure 1.7), the patterns are actually organized into a network structure, a portion of which is shown in Figure 1.9. The network structure (most of which is a tree) allows the initial portions of similar patterns to be merged; at the point where two patterns differ, the network branches into separate paths. The ends of these paths point to the appropriate actions.

The network portion of which Figure 1.9 is a part, is called the SENTENCE network. It guides the overall processing and calls the various subnets as specialists on various phrase types. Currently the entire network consists of approximately 700 states.

While this report was in preparation, we began testing a new idea, which allows us to achieve substantial simplification of the SENTENCE network. As things are now, nearly all variations of phrase orderings possible for expressing a request must be stored explicitly in the sentence network (time phrases are an exception). This means, for example, that there must be separate patterns for active and passive forms of each request.

We conjectured that a set of context register values may uniquely specify an action, independent of the order of the phrases in the requests. If this conjecture were true, we could replace all the ordered patterns which point to a single action with a single unordered pattern. In implementation terms, it means that we can replace the SENTENCE network with (1) a simple program to parse all the phrases in a request, and (2) a table of pattern-action pairs, where each "pattern" is a specific set of context registers with values.

In addition, this would also greatly simplify extending the program within our domain--each pattern/action pair would only need to be coded once, and would thenceforth handle many variations for expressing a request. If the conjecture is true, it will (1) show that natural language requests are redundant, and show (2) that it is possible to exploit this redundancy to make our system simpler and easier to extend. Our preliminary results indicate that the conjecture is true.

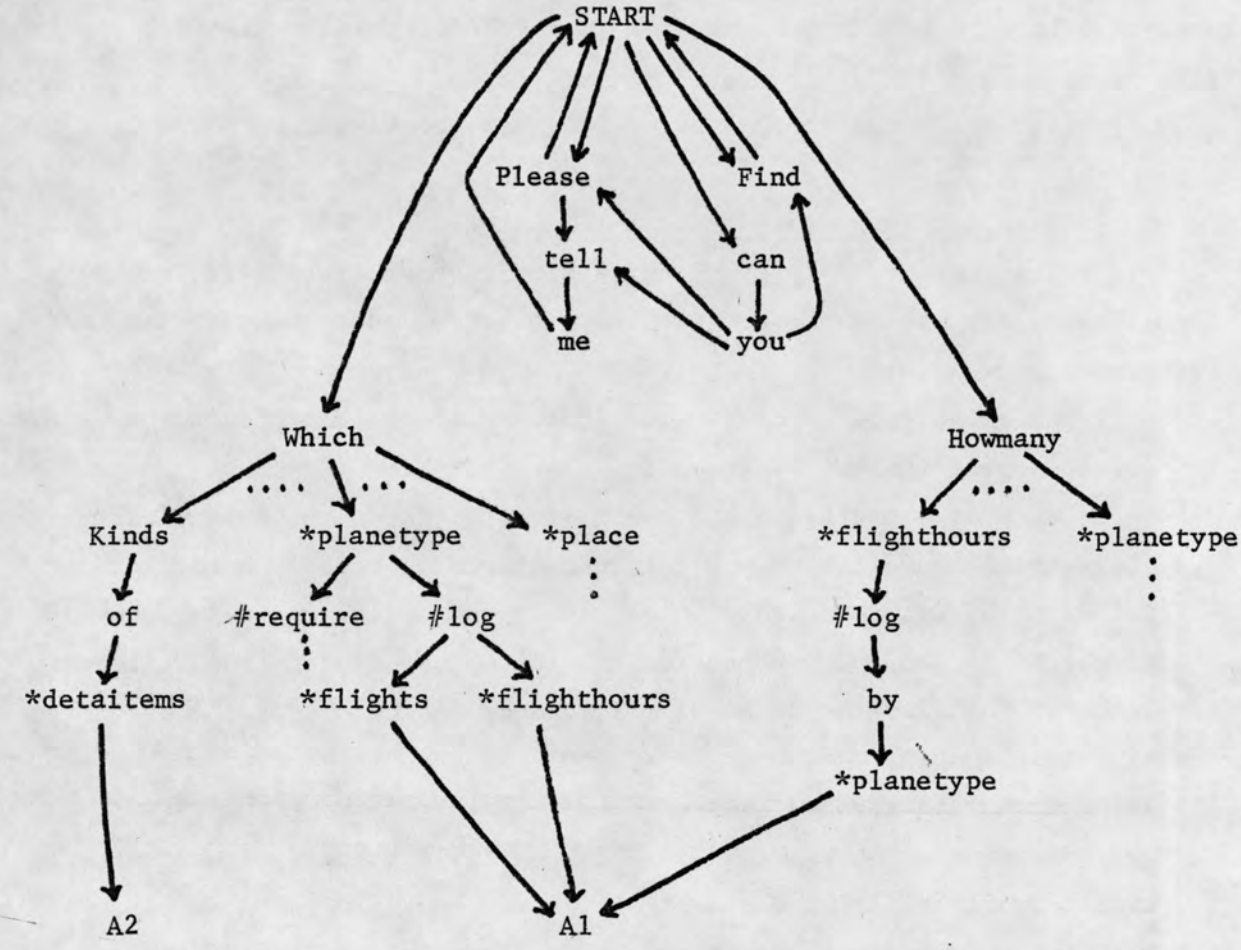


Figure 1.9 Simplified portion of the prestored request network.

We are currently working out details for handling ellipsis and pronoun reference (which must be done somewhat differently) and for dealing with requests involving conjunctions (e.g. or, and, but) and comparatives (e.g. greater than, more than), using this new processing model.

1.3.5 Translation Into Query Language

The paraphrase language expression is next translated into a formal query expression for use with our relational data base system. The translation involves:

(1) Deciding which card types to look at to retrieve the information necessary for answering the user's request;

(2) Deciding what data fields to return from the cards which are searched. (In general, more fields are returned than are actually asked for. For example, if asked about which planes had engine maintenance during some time period, the system returns not only the plane identification numbers, but also the dates of maintenance and codes for the exact type of maintenance.);

(3) Deciding how to arrange the output data. Typical orderings are by increasing or decreasing size of some field value (like number of hours down time) or sequentially by date.

(4) Deciding which operations should be performed on the fields returned. Examples of operations include list, count, average, sum, and find largest;

(5) Translating field values (e.g. for dates, plane types, or actions) into internal data base codes.

The translator produces an expression in the relational calculus [Codd 1971b, Date 1975] which can be used to implement the actual data base search.

Example 1.3.4

Following through with our example, the translator changes the paraphrase language form given in Figure 1.8 to the query language (relational calculus) expression given in Figure 1.10.

```
(FIND  'ALL
       '((V 0))
       '((V BUSER (V TOTHRS))
       '(AND (EQU (V ACTDATEYR 3.)
             (EQU (V ACTDATEMON 1.)
             (EQU (V PLANETYPE) 'A7)
             (LT (V TOTHRS) 20.)
             (GT (V TOTHRS) 10.))
       '(BUSER  UP))
```

Figure 1.10 Query language expression representing Q1.4.1, obtained from paraphrase expression, shown in Figure 1.8.

The meaning of this expression is as follows:

(1) "FIND 'ALL" means that all items satisfying the criteria following should be searched for. In contrast, "FIND 'ONE" would return as soon as any item satisfied the rest of the query expression. Still other forms are possible.

(2) V is a variable name;

(3) 0 specifies a relation. In this case the relation 0 is in turn associated with the data base files in which daily flight data summaries are stored;

(4) '((V BUSER) (V TOTHRS)) specifies what is to be returned as output, in this case a table with columns for BUSER (BUreau SERial number - a plane identification number) and for TOTHRS, the total number of flight hours for that plane;

(5) The five lines beginning with '(AND... specify the predicate which must be true of each entry recorded in the answer table, namely that the year must be 3 (1973), the month must be 1 (January), the plane type must be A7, and the total number of flight hours (given in the card field TOTHRS) must be less than 20 but greater than 10.

(6) '(BUSER UP) means that the output table is to be ordered by BUSER number, with the smallest BUSER number first and the largest BUSER number last.

1.3.6 Execution of Query Expression on the Relational Data Base

In the relational model [Codd (1970, 1971a), Date (1975)] data is viewed as being divided into relations which correspond to files or sets of files in conventional data base terminology. Each relation contains a collection of tuples which correspond to records; each tuple contains one or more domains or fields. A relation can conveniently be thought of as a table, with each row being a tuple and each column a domain.

For our purposes the relational approach has two important advantages:

(1) The relational approach stresses data independence. This means that the user and front end programs are effectively isolated from the actual data base organization. We are now working with only a small subset of the entire 3-M data base; if we were to use our front end with the entire 3-M data base, the data accessing programs would have to be modified, but, using the relational model, these changes need not affect the "data model" seen by users and the natural language front end.

(2) The 3-M data base is already internally organized in a tabular form, and is thus ideally suited to a relational data model.

In the last section, we showed the sort of expression generated by the translator. This expression is given in the data sublanguage Alpha (DSL Alpha [Codd, 1971b)], as implemented in LISP by Fred Green (1976). This expression is used by the relational data base system to construct the actual program used to retrieve the data, along lines suggested in Palermo (1975). In order to construct the search program, the system must:

(1) Select the files to be searched;

(2) Select an order for searching these files;

(3) Generate an expression for testing and selecting tuples values to return while searching;

(4) Generate a program to combine data, possibly from a number of different relations, so that the proper answer will be returned.

(5) Decide when to save the results of a search for future use. This is important in interactive querying, since interesting results can be expected to evoke follow-up queries from a user, and such a query is likely to reference tuples just retrieved.

(In this chapter we will deal only with the first three items above; see chapter 3 of this report for more details on the other two portions of the system.)

Our data base is physically organized into files, where each file contains one type of data (e.g. flight data, failed parts, installed parts, etc.) for a single plane for a one year period. Each file also contains statistical information (including the number of unique values of a domain, the range of values for a numerical domain, and the number of tuples in the file) which is used in planning the order for searching files (step (2) above). Continuing our example should help in understanding this portion of the processing.

Example 1.3.5

By looking at the expression given in Figure 1.10, the execution program first finds the files to be searched. In this case the system will select flight data files for all A7s for the year 1973. These are selected by noting that only relation 0 (flight data summary) is selected, that only data on PLANETYPE equal to A7 will be returned, and that only ACTDATEYR 3 (1973) should be considered. The system notes that all files belong to the same relation, so search order is irrelevant, since no intermediate results need to be generated and combined. Since we have already guaranteed that the files retrieved will contain only data on A7s during 1973, we can simplify the function used to test each tuple (record) to:

```
(AND (EQU (V ACTDATEMON) 1.)
     (LT (V TOTHRS) 20.)
     (GT (V TOTHRS) 10.)).
```

Since we are only to return a list, and the files already contain summaries of flight hours by month, the returned data form is also particularly

simple, requiring no arithmetic operations (any "hits" are merely listed).

1.3.7 Returning Answers

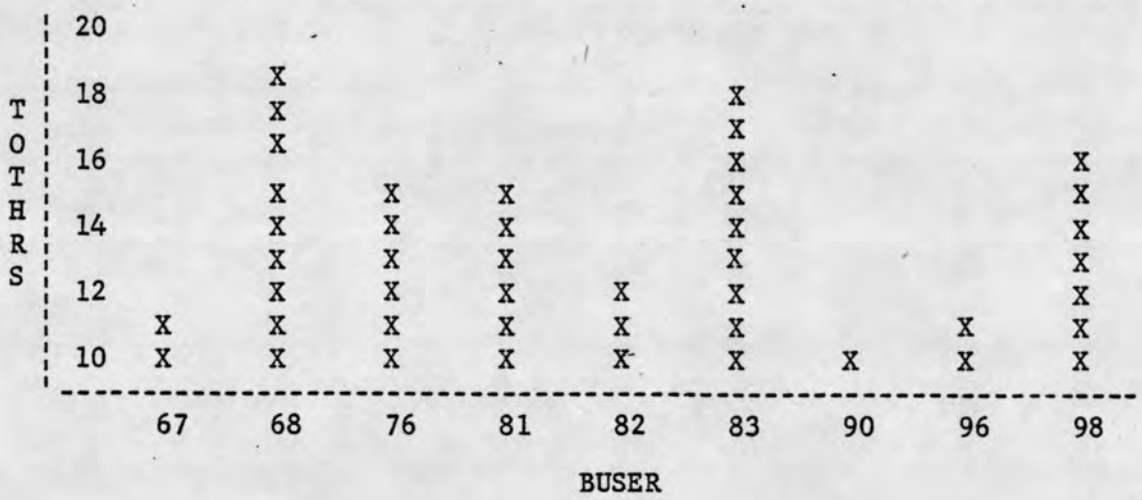
Once the data has been retrieved, the results are passed to an output module, which decides on an appropriate display format for the data. If possible it attempts to produce a graph. This can only be done if (1) pairs of items are returned, (2) one item is numerical, (3) the number of items returned is small enough (but not too small) to produce a reasonable graph which will fit on a CRT screen. If a graph is not possible, the system will produce a list or table; if there is too much data to fit on the screen, the results will be automatically output to the line pointer.

Example 1.3.6

Suppose that only a very few planes flew between 10 and 20 hours in January 1973. A typical output would be:

BUSER	TOTHRS
67	11
68	18
76	14
81	14

A larger number of items could result in a graph, such as:



If 40 or more items were returned (impossible for this query with our current data base, since we have data on only 36 A7s) the results would be output by line printer. The heuristics for selecting output can be easily modified to produce whatever form of output is desired.

1.4 Other Work and Summary

Various portions of this system are treated in greater detail in the rest of this report. In addition, work underway on answering complex and vague questions, and on "browsing" (looking for "interesting" data in a relatively non-directed manner) and "alerting" (screening data as it is input for material deemed "intereating" by users) is described in Chapter 5, and in greater detail in Conrad (1976).

The rest of this report is organized as follows:

Chapter 2 describes the language understanding phase in detail.

Chapter 3 describes our relational data base system and also gives background information on other such systems.

Chapter 4 describes various components of the system and implementation details.

Chapter 5 is a preliminary report on our work on browsing.

Appendices describe the 3-M data base, the kinds of questions that the Navy has asked of this data base in the past, and other miscellaneous information.

2. The Language Understanding Phase

2.1 The Network

The parsing is accomplished by a large ATN network. The network consists of a SENTENCE network and twelve specialized sub-networks which recognize such constituents as time specifications, plane specifications, etc. The entire parser currently consists of approximately 700 states.

The SENTENCE network guides the overall processing of the user's request. It attempts to ascertain what basic kind of question the user is asking and applies the various specialists to gather the constituents. When it has processed the entire sentence, it constructs an internal representation in the Paraphrase Language (see section 2.3).

2.1.1 The Specialist Sub-Networks

Whenever the SENTENCE network suspects that a particular type of phrase is present in the user's request, it calls the appropriate specialist sub-network. The specialist attempts to parse that constituent, and, if successful, returns a representation of what it found in the paraphrase language.

The following gives a brief description of the twelve specialist networks and some example of typical phrases they recognize.

PLANETYPE: Recognizes noun phrases which refer to a plane or group of planes. Examples of the phrases it recognizes are:

"Skyhawks that had engine damage"
"The F4 with tail number 00048"
"Plane 061"
"BUSER number 016"

MAINTTYPE: Recognizes phrases referring to maintenance, either by their work-unit-code, action-taken-code, or type-maintenance-code. Some of the constructions recognized are:

"unscheduled maintenance on ..."
"scheduled work with ..."

MAINTAT: Recognizes maintenance actions when referred to by their action-taken codes. Some examples of these phrases are:

"action taken code Q"

"AT 6"

MAINTTM: Recognizes maintenance actions when specified by a legal type-maintenance code. For example:

"TM Y"

"type maintenance code 10"

MAINTWUC: Recognizes maintenance when referred to by a legal work-unit code. Examples of accepted phrases are:

"system work unit code 46"

"WUC 91"

"work unit 97"

DAMAGETYPE: Recognizes a specification of damage to aircraft. The damage can be to a general area or system, or can refer to a particular how-malfunction code or codes. For example:

"damage to electronic systems"

"damage due to howmal code 53"

"launch damage"

"mechanical systems damage"

"HOWMAL 45 damage"

TIMEPP: Recognizes specifications of time periods, either as prepositional phrases or other adverbial phrases. A time period can be a day, a month, a year, or a span of time between two days, months, or years. For example, the following are accepted:

"last month"

"before April 73"

"between January 1 and June 31, 1975"

"during the month of August"

"on 10 Dec. '75"

DATE: Recognizes a time period which is a date. A date can contain a value for the month, day and year. Some strings which are accepted include:

"June 29 '73"
"1 January"
"July fourth, 1976"
"1 2 76"

PLACE: Recognizes a phrase denoting a location. Our current implementation accepts locations in a coded form, although a few common names are included. Some examples are:

"P9R"
"Midway"
"VMFA-122"

CODETYPES: Recognizes phrases which refer to one of several classes of codes used in the data base. These include the action-taken codes, work-unit codes, etc. Examples are:

"when discovered codes"
"malfunction codes"

ABBREV: Recognizes a number of phrases which are mapped into the appropriate abbreviations as well as accepting abbreviations and producing the appropriate phrase.

AMOUNT: Recognizes phrases used as numerical quantifiers. It produces a list of predicates which represent the phrase. Examples are:

"three or more times"
"more than 6"
"between 100 and 200"
"greater than 100 but less than 1000"
"30"
"any"

2.1.2 An Example of a Network

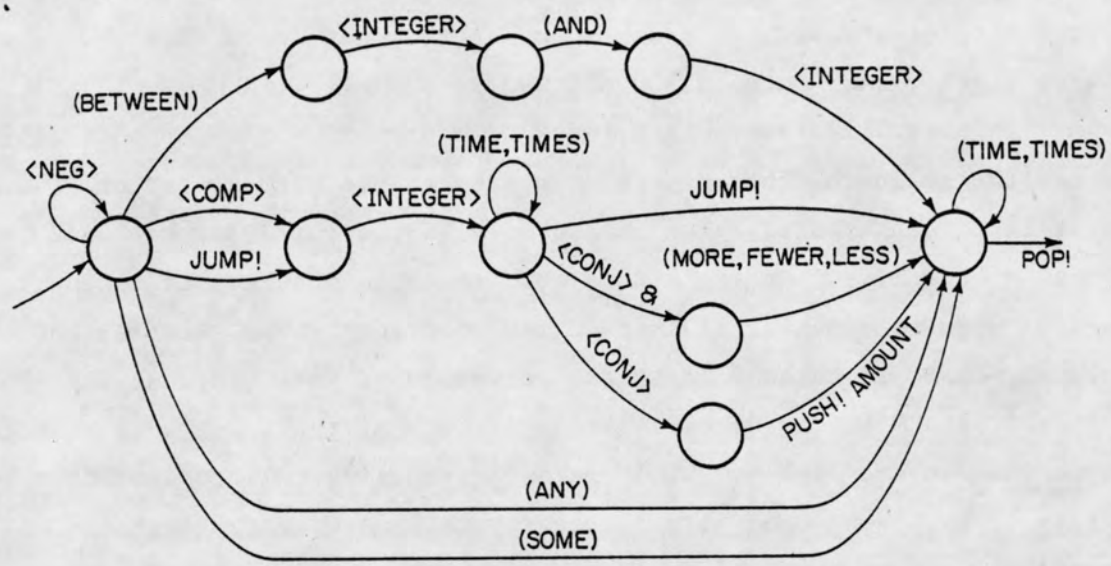
The AMOUNT net is delegated the task of recognizing phrases which are used as numerical quantifiers and producing a list of predicates which represents the phrase. It accepts such strings as "more than 6", "exactly twelve times", and "between 100 and 200".

Figure 2.1 shows this network in a graphic form. In this figure we use the following conventions. An arc labeled with a word or list of words in parentheses is taken if and only if the current input word is a member of the list. An arc with a word in angle brackets is taken if the current word is a member of that lexical category. For example, the arc labeled <CONJ> is taken only if the current word is a conjunction. An arc labeled JUMP! can be taken at any time, but the input word is not advanced. An arc labeled PUSH! makes a recursive call to another subnet before it is taken. The actual code for this network is given in Appendix C.

2.2 The Parser

Although we call this stage parsing, the term is somewhat misleading. Conventionally, a parser is an instantiation of a grammar. A grammar is a body of syntactic knowledge which is concerned with the structure of well-formed sentences (or strings) and not their meaning. We have chosen to implement a "semantic grammar" [Brown and Burton (1976)], that is, a grammar which encorporates both syntactic and semantic information. This grammar will accept only those sentences which are semantically well formed (i.e. meaningful) with respect to our system.

Thus, in our system there is no real distinction between knowledge which is syntactic and knowledge which is semantic or pragmatic. For example, the knowledge of the general form of a noun phrase (see [Woods (1970)] or [Winograd (1972)]) is not to be found in any one place, but distributed within the various semantic noun phrase recognizers. Our system has such specialists for recognizing noun phrases referring to planes, maintenance actions, codes, times, locations, etc.



<COMP>	<CONJ>	<INTEGER>	<NEG>
MORE THAN	AND	1,2,3...	NO
AT LEAST	BUT	ONE	NONE
EXACTLY	OR	TWO	NOT
APPROXIMATELY		THREE	
AS MANY AS			

FP-5042

Figure 2.1 The AMOUNT net

2.2.1 Advantages of a Semantic Grammar

We have found that the notion of a "semantic grammar" has a number of important advantages. Some of these are:

(1) Sentences which are parsed are only those which are "meaningful" to the system. For example, the sentence:

"How many engine repairs required maintenance in May"

is immediately rejected. Our parser embodies the concept of 'plane require maintenance' but not that of 'maintenance require maintenance'. Because a sentence must closely fit one of a number of 'request frames', the chance of it being misunderstood is lessened.

(2) The processing of a sentence is guided by syntactic, semantic, and pragmatic knowledge. We can make use of predictions and expectations to focus on the 'most likely' interpretation of the sentence. Many potential ambiguities can be immediately resolved or never seen at all. For example, the word WING is ambiguous between the sense "a part of a plane" and the sense "a squadron of aircraft". The local context of a sentence is often all that is necessary to resolve the ambiguity. The sentence:

"Were repairs done to that wing?"

selects the first sense since we recognize the concept 'someone repair plane-part' but not that of 'someone repair plane-group'.

(3) We can achieve efficient recognition of constituents through the use of specialists. A general noun phrase parser, such as the one used in the LUNAR system [Woods et al, (1972)], must be aware of many syntactic constructions which are special to certain constituents. For example, most of the syntactic features which can appear in a general noun phrase will <u>not</u> be found in a noun phrase which is specifying a date or a time. Some syntactic forms are closely tied to certain words or semantic classes of words. For example, "factive" nouns such as FACT, IDEA, MOTION, etc., can be used to introduce a non-reduced that-clause as in sentence (S1).

(S1) The fact <u>that</u> <u>the</u> <u>plane</u> <u>crashed</u> is important.

(4) The use of a semantic parser simplifies the handling of the problems of ellipsis and anaphoric reference. When our system discovers a pronoun or hypothesizes that a constituent has been deleted, it can immediately restrict the set of possible referents on semantic grounds. The referent must be one which satisfies the semantic constraints of the sentence as seen so far. Consider sentence (S2) when followed by either (S3) or (S4).

(S2) "Which Skyhawk suffered birdstrike damage?"

(S3) "Did it suffer any other damage?"

(S4) "When did it occur?"

In sentence (S3) the "it" can only refer to the skyhawk of sentence (S2) since the parser recognizes the concept of 'plane suffer damage' but not that of 'damage suffer damage'. Similarly, in processing sentence (S4), the only candidate referent is "birdstrike damage" since the parser accepts sentences of the form "When did <damage> occur" but not "When did <plane> occur".

(5) Treating syntactic and semantic knowledge in a more uniform way leads to a well-balanced system. It is our observation that most natural language processing systems tend to have much more developed syntactic processors than semantic processors. In other words, they can parse more than they can understand. Although this is not a serious drawback, it can cause considerable overhead in a practical system. In addition, many non-grammatical requests can be correctly understood by a semantic parser which would be rejected on syntactic grounds by a system which first performed a syntactic parse on all input.

2.2.2 Disadvantages of a Semantic Grammar

We feel that our current system has worked quite well at the present level of competence. As we try to extend the domain of discourse of the system, however, the basic structure will have to be rethought. It is our belief that to achieve a truly significant level of intelligent linguistic performance we must more fully investigate the interactions between the different kinds of linguistic knowledge.

(1) The primary difficulty with a "semantic grammar" is that it does not capture much of the regularity of natural language. A semantic grammar is specifically designed to be conversant in a single domain of discourse. If we wish to construct a similar system for a different domain, there is no clear package of syntactic knowledge that the two can share. This problem exists for other kinds of "linguistic" knowledge as well. For example, every natural language system should have knowledge of the pragmatics of conversation. When a user says "I want to see X", he is not just informing the system of his desires, but commanding it to "show him X". Such a knowledge of speech acts is independent of the domain of discourse. Indeed, this sort of knowledge seems to be a language universal which cuts across languages and cultures.

(2) The fact that we are treating syntax, semantics, and pragmatics on a uniform basis has other effects. If a sentence is rejected by the grammar it is difficult to discover where the problem lies. It is not possible to distinguish sentences which are ill-formed syntactically from those ill-formed semantically or pragmatically. This situation has an impact on the ease with which such a system can be modified, either by the creators or itself.

(3) The ability of a system to apply its syntactic knowledge in the absence of a semantic component would allow it to more easily handle certain kinds of learning. For example, consider the sentence:

(S5) "Did any of these hanger queens fly last month?"

Let's suppose that our system does not know what the phrase "hanger queen" means. Given this sentence, it could not progress beyond that phrase and would be unable to use the latter part of the sentence to help narrow the possible meanings. We can imagine a system which could, in this case, apply syntactic and morphological specialists to recognize that "these hanger queens" appears to be a well formed noun phrase, and could then continue to process the sentence.

If the other components of the sentence are recognized, it would not be difficult to hypotehsize that a "hanger queen" might refer to an aircraft. With this hypothesis in hand, the system would be in a

good position to interact with the user and perhaps nail down an adequate notion of what is meant. (We, in fact, intend to add this ability to our system in the near future.)

(4) A potential problem with the approach we have taken is the strong "top-down" nature of our recognition strategy (as opposed to "bottom-up" or data driven). In our current system, we feel that the advantages of a top-down approach clearly outweigh the drawbacks. A straightforward extension to a considerably larger domain of discourse may reverse this relationship.

2.3 Context Registers and Subnets

In the PLANES system we have tried to combine as many of the syntactic and semantic processes together as possible. We felt that as the parsing task occurs, useful semantic information is also being generated concurrently. To gain access to this information, much of our parser was concerned with parsing specific entities known to be required for inquiries about information in the PLANES data base. Such necessary information includes the types of planes, the damage types, the maintenance types, the time period in question, etc. As each of these entities is matched, it is saved--either in some coded form or exactly as it appears in the sentence--in a register. We call these registers "context registers."

The context registers are normally filled with the information that has been returned from the subnets of the augmented transition network. These subnets are designed to parse specific parts of the user's request. When a parse of a specific entity occurs in the subnet, a list is generated that contains the "essense" of the entity. Some of the information coded in the lists includes such information as whether or not a pronoun was parsed, specific code numbers, or dates. A list of subnets and what they return to the upper level net (a process called "popping") follows.

*planetype: Parses a name of a plane. It pops up

```
(plane (pronoun +) (type +) (buser +)
       (plneg +) (pldam +) (plmai +)).
```

*mainttype: Parse a maintenance-type, maintenance-action, or work unit code. It pops up

```
(maint (pronoun +) (type-maint +)
       (action-taken +) (workunitcode +)).
```

*maintat: Parse an action-taken(at) code that gives the current status of a maintenance task. It pops up (⟨at⟩).

*mainttm: Parse a maintenance type(tm) code. It pops up (⟨tm⟩).

*mainwuc: Parse a work unit code(wuc). It pops up (⟨wucsys⟩).

*damagetype: Parse a how malfunctioned code(howmal). It pops up (⟨howmal⟩).

*timepp: Parse a time range. It pops up

(time ⟨date 1⟩ ⟨date 2⟩).

*date: Parse a date. It pops up

(date (month +) (day +) (year +)).

*place: Parse a place. It pops up (place +).

*codetypes: Parse a codetype (describes the information available in the data base). It pops up (⟨code⟩).

*abbrev: Parse an abbreviation or a phrase to be abbreviated. It pops up

(abbrev ⟨phrase⟩ ⟨abbrev⟩).

*amount: Parse a quantifier. It pops up

((⟨relation⟩ ⟨no.⟩) (⟨relation⟩ ⟨no.⟩)).

The information that is popped from the subnets is stored directly in the context registers. These registers can then be used to form a paraphrase of the user's request. The paraphrase generated by our current system is somewhere in between the user's English request and the formal query language needed to search the data base. Eventually we intend to generate a more English-like paraphrase. It is a relatively simple matter

to convert the paraphrase into the query language. Further details can be found in the section on the Translator (Section 4.3).

The paraphrase currently popped for each sentence is of the form:

(OPER (PLOT ⟨T or NIL⟩) ⟨SUBJ⟩ ⟨QUANTS⟩ (NEG ⟨T or NIL⟩)
(⟨ACT⟩ ⟨OBJ⟩ ⟨QUANTO⟩) ⟨TIME⟩ (⟨ORIGIN⟩ ⟨DESTINATION⟩))

OPER: count, findall, findone, dates, where

PLOT: T if want answer plotted; NIL otherwise

SUBJ: *planetype, *mainttype, *damagetype, time, hours, flights, failures, parts

QUANTS: Contains any quantifier of the subject; e.g. ⟩20 flights, ⟨15 hours, etc.

NEG: T for negation of the action; NIL otherwise

ACT: malfunction, occur, repaired, on, arrive, nor, nors, norms, normu, fly, found, need, perform [1]

OBJ: same as subject

QUANTO: contains any quantifier of the object

TIME: gives the time period over which the question is to be answered

ORIGIN: the point of origination of a flight

---

[1] NOR = not operationally ready; +S = due to supply (waiting for parts); +MS = due to scheduled maintenance; +MU = due to unscheduled maintenance.

DESTINATION: the destination of a flight

As an example, the paraphrase generated for "Where was plane 3 repaired between November 1 and December 30 1972?" is:

```
((⟨OPER⟩  WHERE
 ⟨PLOT⟩  (plot NIL)
 ⟨SUBJ⟩  (plane (pronoun NIL)
                (type PLANE)
                (buser 3.)
                (plneg NIL)
                (pldam NIL)
                (plmai NIL))
 ⟨QUANTS⟩  NIL
 ⟨NEG⟩     (neg NIL)
           (
 ⟨ACT⟩      REPAIRED
 ⟨OBJ⟩      NIL
 ⟨QUANTO⟩   NIL)
 ⟨TIME⟩    (time (date (month (11. 304. 305.))
                       (day 1.) (year 72.))
           (date (month (12. 334. 335.))
                 (day 30.) (year 72.)))
 ⟨ORIGIN⟩  (NIL
 ⟨DESTINATION⟩   NIL))
```

2.4 Pronoun Reference and Ellipsis

Pronoun reference is one of the more difficult tasks in natural language understanding. It requires using the current context of a sentence to find a set of possible elements that the pronoun may stand for from the immediate past history of inquiries to the system. After weighing the merits of each member of the set, the system must select one as the "best" choice to replace the pronoun. Ellipsis is a very similar problem that occurs when a request does not express a complete thought. It involves filling in missing information from a user's request with information from the immediate past. As an example a user might follow the question "How many maintenance actions did plane 004 have in May 1974?" with "In June?".

Winograd (1972) handles pronoun reference at the time a pronoun is parsed. He utilizes a set of programs that are designed to handle specific pronouns. For example, the definitions of "it" and "they" utilize a special heuristic program which looks into the past for anything that the pronoun might refer to. For each possibility, a value is assigned which represents the likelihood that the phrase could be represented by the pronoun. When more than one phrase is possible, all are carried along through the rest of the analysis of the sentence. At the end, an ambiguity mechanism is applied to decide which is the best choice. If no choice can be made, the user is asked for clarification.

Wood's (1972) routine analyzes pronouns in the semantic phase. When a pronoun is found as a node in the syntax tree, a function is applied to the node to try to determine what the pronoun stands for. This determination is made by searching through a list of antecedent noun phrases for one that has a syntactic and semantic structure parallel to the given node. The node is then replaced by the antecedent noun phrase that was selected.

The SOPHIE [Brown and Burton (1975)] natural language processor handles ellipsis in a novel way. When it deduces that information is

missing from the current request, SOPHIE looks for a previously mentioned use of a currently specified object. The semantic grammar is used to identify the type of concept that is involved. A context mechanism then searches the past history for a specialist in a previous parse that will accept the given concept as an argument. Once this is found, the phrase can be substituted into the current request.

Our current implementation of PLANES handles pronoun references to the previous request only. This is accomplished relatively easily and cheaply by simply saving the card images found during the last query of the data base in a temporary file. Thus, when a pronoun is parsed, the query language is generated to search the temporary file only. Thus, for example:

(1) "How many flights did plane 48 make during Dec. 1969?"

Paraphrase generated:

```
(COUNT (plot NIL)
       FLIGHTS
       NIL
       (neg NIL)
       (FLY (plane (pronoun NIL) (type PLANE)
                   (buser 48.)(plneg NIL)
                   (pldam NIL)(plmai NIL))
            NIL)
       (time (date (MONTH (12. 334. 335.))
                   (day NIL) (year 69.))
            NIL)
       (NIL NIL))
```

```
Query language generated:
(FIND 'ALL
      '((V 0))     ["0" is the type of file to search]
      '((SUM (V TOTFLTS)))
      '(AND (EQU (V ACTDATE) 912.)
            (EQU (V BUSER) 48.))
     'NIL)
```

(2) "How many flight hours did it log?"

(Thus, "it" refers to "plane 48" in (1).)

```
Paraphrase generated:
(COUNT (plot NIL)
       TIME
       NIL
       (neg NIL)
       (FLY (plane (pronoun IT)
                   (type NIL) (buser NIL)
                   (plneg NIL) (pldam NIL)
                   (plmai NIL))
            NIL)
       (time (date (month (12. 334. 335.))
             (day NIL) (year 69.))
             NIL)
       (NIL NIL))
```

Query language generated:

```
(FIND 'ALL
      '((V 00031))   [this causes a search of only the
                      temporary file 00031 to take
                      place.]
      '((SUM (V TOTHRS)))
      '(AND (EQU (V ACTDATE) 912.)
            (EQU (V BUSER) 48.))
      'NIL)
```

One can see that the use of temporary files in handling pronoun references can be very efficient since it saves searching through a great number of cards that had been previously searched. However two limitations occur in the current implementation: (1) pronouns can only reference noun groups in the previous request; and (2) should a request use a pronoun reference to the last sentence but ask for information that is contained in a different set of cards than those in the temporary file, no answer could be found. As an example, consider

(1) "How many flights did plane 48 make in April 1973?"

This would form a temporary file of the monthly summary card for April 1973 for plane 48.

If we followed with:

(2) "What damage occurred to it?"

This could not be answered since the temporary file <u>does not</u> contain damage codes.

We wish to solve these problems without giving up the time advantage introduced by the temporary files. It should be relatively easy to solve the second problem. We can simply construct a table of fields present on each type of data card. Then if the field we must key on due to the user's request is not present on the cards in the temporary file, we will know that we must search the data base directly.

The solution to the first problem will be somewhat harder. It will require the saving of the past history of all paraphrases, queries, and temporary files. These can be saved in a stack (possibly of fixed size to limit its size) to allow us to examine the most recent entries first. To hypothesize what the pronoun refers to, we must decide what category it is applicable to (simply examine the current paraphrase for this), and then find the last non-pronoun entry that was made in this category. Further examination of the paraphrase and its query language request can be used to either reinforce or deny our hypothesis. If too many conflicts appear, we may search further back, otherwise we will assume that we have found the correct pronoun reference. When the conflicts cannot be resolved, we can ask the user to tell us which one of a number of possible choices the pronoun stands for. Once a reference point has been found to an object in the past, we can examine all temporary files back to that point to see if they can be used to answer our request. If not, we can still search the data base directly.

This method will also give us another advantage. We should be able to change the current context back to an earlier environment; e.g. by stating:

"Earlier we were talking about Skyhawks."

We can search back through the paraphrases, query requests, and temporary files for a reference to "Skyhawks," and then be ready to use previous information over again.

Storing the context registers in a stack makes the saving of previous paraphrases relatively easy. It also makes ellipses relatively simple to handle. We simply fill in any missing--but required--information by finding the last non-empty entry of a particular context register. Thus if given the sentences:

(1) "Tell me all the malfunctions found on the A7 with buser 12 on Feb. 15, 1970."

(2) "On Feb. 16, 1970."

the syetem can generate a paraphrase for the second request by copying all context registers from the last request except the one for the time period. The new time period is then used. As another example, if given

(3) "How many A7's had greater than 20 NOR hours on Feb. 15, 1970?" and

(4) "How many had greater than 30 NOR hours?"

all context regesters for the first request are copied except the ones used to quantify NOR hours. This number is filled in from the new request.

The detection of missing information occurs by determining if a sentence has run out of words permaturely or if a verb appears where a noun phrase is expected. The beauty of the transition network is that even if a mistaken ellipsis is made, the parsing process continues, and the mistake will most likely be detected further into the parsing process.

2.5 NETEDI: The Augmented Transition Network Editor

The structure of our augmented transition network (ATN) [Woods (1970)] is such that we often must add to it new sentence types. Before NETEDI existed, we had three ways of doing this: the TECO or SOS standard languages for editing files, and the LISP editor [Gabriel and Finin (1975)].

It turned out that most of the new arcs we added for these new sentence-types were "WRD" arcs. Here is an example of a typical WRD arc:

```
(WRD WHAT T (TO S:12))
```

This recognizes the word "what" and jumps to state s:12. (The "T" is a test; any LISP predicate could be inserted in place of it. If the predicate evaluates to "T" the arc is taken.)

An arc similar to this one has to be inserted for each work in a new sentence-type. As one can perhaps guess, this is a tedious process. Furthermore, it is not really necessary. Since most of the arcs will be of a similar form, why not let the computer do the busy work? All we should have to tell it is which words we want recognized and in what order. A program can then make up a next state, form the arc and insert it into the ATN.

Similar redundancies appear in other types of arcs. NETEDI has been designed to rid the programmer of the necessity of contending with these and also to give him or her a concise, easily understood representation of sentences to be added to the ATN.

Here is an example of a sentence-form:

```
((WHAT WHICH) *PLANETYPE ($ROOT (NEED REQUIRE))
  *MAINTTYPE IN MAY)
```

This is the kind of input that NETEDI receives and turns into new parts of the ATN. Notice that the sentence-form is in the form of a list. NETEDI looks at elements of this list one at a time. We'll follow this example through NETEDI assuming we have an ATN as in figure 2.2. (The real ATN is much larger, of course.)

Basically NETEDI is composed of three stages: (1) the tracing stage, (2) the creating stage and (3) the answer stage. During the tracing stage NETEDI tries to match as much of the sentence-form as it can to existing arcs in the ATN. During this stage the ATN is not changed at all-merely examined.

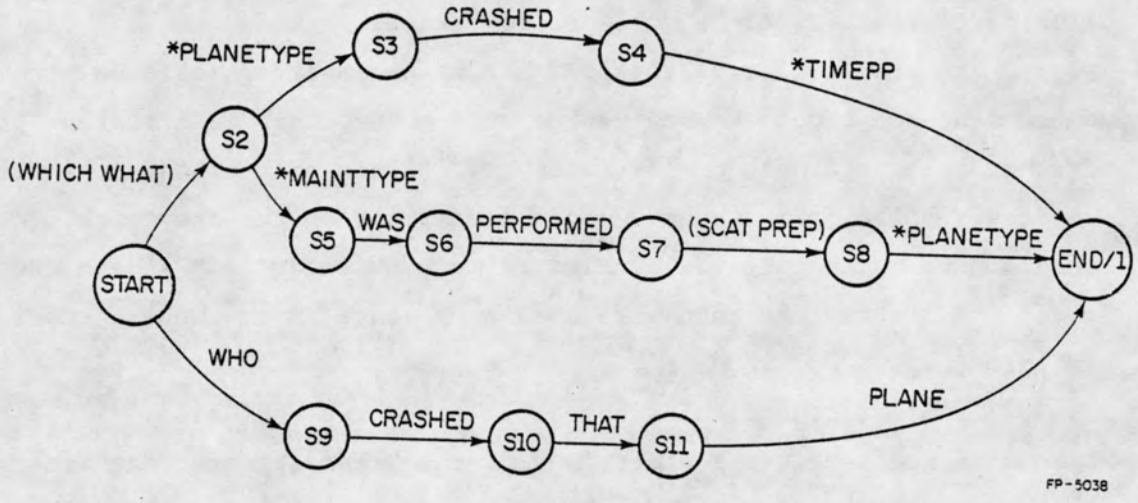


Figure 2.2 An ATN for NETEDI operation example

The first thing NETEDI sees in the sentence-form is the list "(WHAT WHICH)". Since it's in the tracing stage, it tries to find an identical arc pointing out of the START state. There is one: it goes to state S2. This is called a "WRD arc". Now NETEDI looks at state S2 hoping to find an arc to match "*PLANETYPE" (a "PUSH arc"). Again it is successful: it finds the arc which points to S3. At S3 NETEDI cannot find an arc to match the next element in the sentence-form, i.e. "($ROOT (NEED REQUIRE))", so it shifts into the creating stage.

It should be mentioned that when NETEDI sees in its sentence-form a single word not starting with "*", it doesn't just proceed to look for an associated WRD arc; before doing so, it checks its dictionary for two things:

(1) It tests for compound words whose parts we would rather treat separately. For instance, "anything" is treated by PLANES as the two words "any" and "thing". If this information is entered in the dictionary for a word and NETEDI encounters that word in one of its sentence forms, it is able to make the necessary substitution. So the sentence-form "(DID ANYTHING CRASH THIS MONTH)" becomes "(DID ANY THING CRASH THIS MONTH)".

(2) It tests for words which are part of common phrases, e.g. "as well as" or "all right". A mechanism exists in PLANES to treat these phrases as one word. This is done by storing the information in the dictionary entry of the first word of the phrase. If it can, NETEDI replaces the phrase with the given single word. For instance the phrase "as well as" gets replaced by the word "aswellas" in the sentence-form and NETEDI tries again. So the sentence-form "(DID A PHANTOM AS WELL AS AN F4 CRASH IN MAY)" is replaced by "(DID A PHANTOM ASWELLAS AN F4 CRASH IN MAY).

The creating stage begins when NETEDI finds that it cannot match an element in the sentence-form to an arc in the net. From here on the program has to create new states and new arcs in the ATN corresponding to the words in the remaining part of the sentence-form.

"Creating" refers to the new arcs and new states which are created in order to recognize new sentences. In our example, we are in state S3 and we have to add an arc to recognize ($ROOT (NEED REQUIRE)). (This arc incidentally, will recognize any word whose root form is "need" or "require", e.g. "needed", "needs", "will require", etc.)

The first thing we need is a name for a new state--one which has not yet been used. This is so the arc will have a place to point to. We can assume that in general arcs begin and end at a state. We can see from figure 2.2 that the name S12 hasn't been used yet. So, we create a state with that name and an arc to it (see figure 2.3a).

This process is continued, adding arcs and states until there are no more words in the sentence-form. When this point is reached, NETEDI, instead of creating a new state for the last arc to point to, connects it to the (already existing) "END" state. The end result of this process is shown in figure 2.3b. The program now moves on to its third and final stage, the answer stage.

The answer stage is named as it is for the following reason: When the system is asked a question, it usually is expected to return an answer, e.g. the question "Which Phantoms required engine maintenance in May?" (one of the questions the example sentence-form is designed to handle) would get an answer something like this: "BUSER 0092, BUSER 0157". The way it forms an answer is to execute a LISP function which searches the data base. The purpose of the answer stage then is to put the appropriate function for the new sentence form at the end of the ATN.

There is now a working version of NETEDI. It can insert any kind of arc, but doesn't know how to do anything else, e.g. set registers, test values, etc. This version also doesn't have a very clean interface to the LISP editor (which it uses as a file manipulator). These problems are being worked on currently.

An example of NETEDI in action follows. In figure 2.4 a sample question is given to PLANES to answer. PLANES can't answer it, so NETEDI is called by the function "CONVERSE" (see figure 2.5). The sentence-form corresponding to the losing sentence (and some others) is typed in

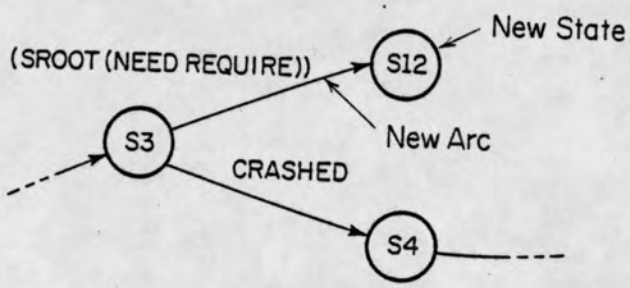


Figure 2.3a Creating a new state and new arc

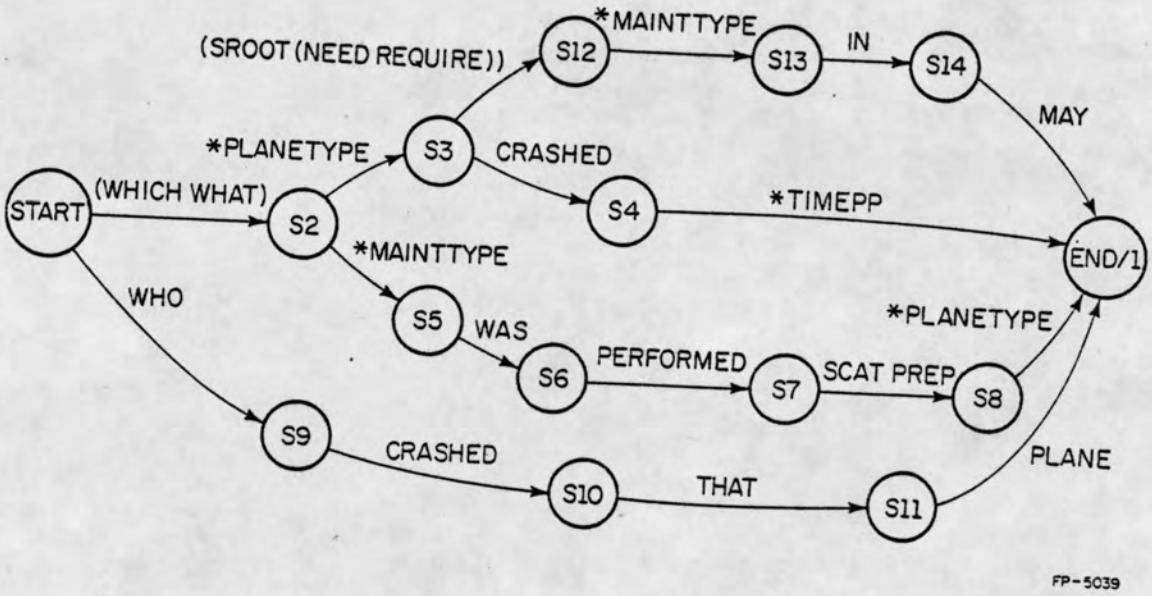


Figure 2.3b Resulting modified ATN (compare figure 2.1)

```
Please enter your question.....
  >> For each month in 1973 show the NOR hours for plane 3

Parsing......
 [CPU time was 0.69 seconds.  Real time was 1.63 seconds.
I could not understand your request please rephrase it.
```

Figure 2.4 NETEDI example; a question PLANES cannot answer initially.

```
(CONVERSE)
AUGMENTED TRANSITION NETWORK EDITOR VERSION 1

PLEASE TYPE THE SENTENCE-FORM TO BE ADDED TO THE NETWORK
(FOR EACH MONTH ($CAT PREP) 1973.  ($ROOT (GIVE SHOW)) THE NOR HOURS
FOR P LANE 3 )

PLEASE TYPE THE ASSOCIATED ANSWER-FORM
(POP (QUOTE (DATES (PLOT) TIME NIL (NEG NIL) (NOR (PLANE (PRONOUN NIL) (
TYPE NIL) (BUSER 3)) NIL) (TIME (DATE (MONTH NIL) (DAY NIL) (YEAR 111))
NIL) (NIL NIL))))

WHAT IS THE INITIAL STATE OF THE NETWORK
S0001
WHAT IS THE FILENAME AND PPN OF THE NETWORK TO BE EDITED?
(PLEASE TYPE IN THE FORM:  ( FILENAME EXT ( PROJECT# PROGRAMER# )) )
(ATNNEW FOO (1000 423))
LOADING:   (ATNNEW FOO DSK (1000 423))
;FASLOADING (ATN FAS DSK (1000 130)) FOR: DEFATN
        16 S-EXPRESSIONS, 0 FUNCTIONS READ.
;FASLOADING (HATN FAS DSK (1000 130)) FOR: EDITATN
;LOADING EDIT.70

;FASLOADING (EDIT FAS SYS) FOR: EDIT1
LOADING EDIT.INI (NIL)
THANK YOU
FOO
14 NEW STATES WERE ADDED TO FOO
NIL?
```

Figure 2.5 Dialogue to add to the system the ability to handle the question from figure 2.4.

along with the associated answer-form, and a few other items. NETEDI thanks the programmer and notes that fourteen states were added to the net.

In figure 2.6 the question is asked again. This time the net can answer it.

Now suppose we want to add to the network the ability to answer a similar question, e.g. one which matches on some initial segment a sentence already in the ATN. Figure 2.7 shows such a question. Note that this time only five new states were added. PLANES can now answer this new type of question, as shown in figure 2.8.

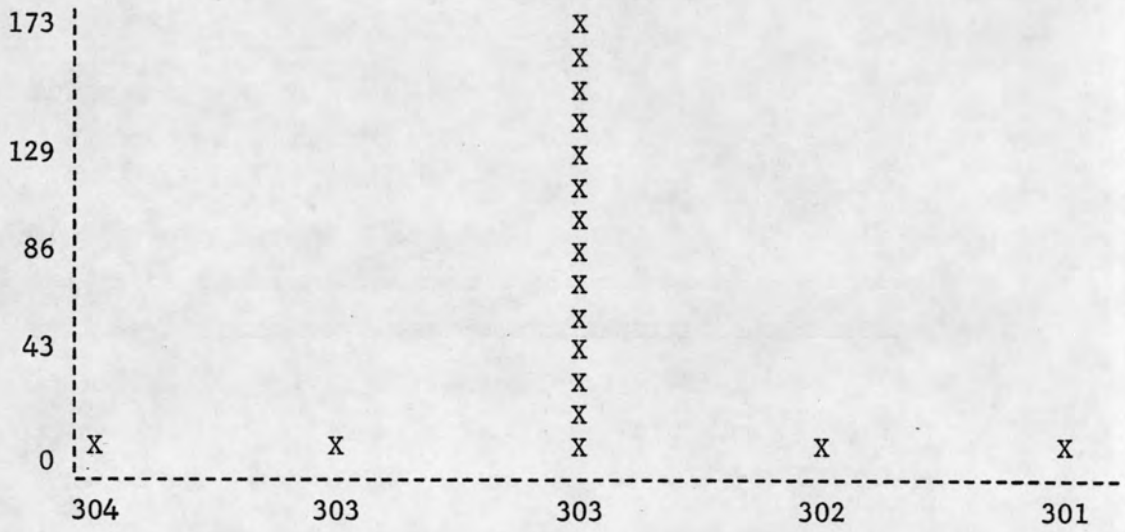
```
Please enter your question.....
  >> For each month in 1973 show the NOR hours for plane 3.

Evaluating.....
 [CPU time was 3.1 seconds.  Real time was 4.73 seconds.

290 |                          X
    |                          X
    |                          X
    |                          X
217 |                          X
    |                          X
    |                          X
145 |                          X
    |                          X
    |                          X
    |                          X
 72 |                          X
    |                          X
  0 | X            X                          X            X
    ------------------------------------------------------------
    304          303          303          302          301

                              ACTDATE

MIN = 0.0              AVERAGE = 58.0              MAX - 290.0
```

Figure 2.6 PLANES can now answer the question.

```
  (CONVERSE)
AUGMENTED TRANSITION NETWORK EDITOR VERSION 1

PLEASE TYPE THE SENTENCE-FORM TO BE ADDED TO THE NETWORK
(FOR EACH MONTH ($CAT PREP) 1973.  ($ROOT (GIVE SHOW)) THE NORS HOURS FOR
PLANE 3)

PLEASE TYPE THE ASSOCIATED ANSWER-FORM
(POP (QUOTE (DATES (PLOT T) TIME NIL (NEG NIL) (NORS (PLANE (PRONOUN NIL
) (TYPE NIL) (BUSER 3)) NIL) (TIME (DATE (MONTH NIL) (DAY NIL) (YEAR 111
)) NIL) (NIL NIL))))

THANK YOU
FOO
5 NEW STATES WERE ADDED TO FOO
NIL
```

Figure 2.7 A related question is now easier to add

```
Please enter your question.....
  >> For each month in 1973 show the NORS hours for plane 3.

Evaluating.....
  [CPU time was 3.0 seconds.  Real time was 6.91 seconds.
```



ACTDATE

MIN = 0.0 AVERAGE = 34.6 MAX = 173.0

Figure 2.8 PLANES can now answer the question from figure 2.7

3. Implementation of a Query Language Based on the Relational Calculus

3.1 Introduction -- The Relational View of Data

The problem with which we have been faced is the development of a query language for the 3-M data base. The intended application of this language is the intermediate language for the PLANES system [Waltz (1976)].

Codd (1970) has introduced the notion of a relational view of data. This Data Model (DM) is discussed in detail by Codd (1971a) and Date (1975). The relational model has its foundation in the mathematical theory of relations. In this model the data is viewed as being divided into "relations" which correspond to files in conventional data base terminology. Each relation contains a collection of "tuples" which correspond to records, and each tuple contains one or more "domains" or fields. A relation can be conveniently viewed as a table with each row being a tuple and each column a domain.

Date regards the relational view as superior to both the Heirarchical view of data (typified by the Information Management System (IMS) [IBM (1971)] and the Network view (typified by the Data Base Task Group System (DBTG) [ACM (1971)] in that the latter two systems are both data dependent while the relational view stresses data independence which means that the user is isolated from the actual physical organization of the data. Data independence is particularly important in this project where we are currently working with a small subset of the large 3-M data base. If our natural language system were to be upgraded to work with the entire data base (or a larger part of it) substantial changes would be necessary in accessing methods and data organization, but, using the relational model, these changes need not effect the "data model" seen by the users. In addition, the internal tabular format of the data base as it is now organized is ideally suited to a relational model of the data. For these reasons, we have decided to use a relational model for the data base.

3.2 A Relational View of the 3-M PLANES Data Base

Having chosen the relational model to represent the data, we were then faced with the problem of applying this model to the 3-M data base. A study of the cardtypes used in the data base showed that certain groups of cardtypes could easily be viewed as relations. The cardtypes thus chosen to comprise each relation are shown in Table 3.1.

In this manner we have organized the data base into a number of relations with each relation containing certain cardtypes. The tuples in a relation now become the individual cards comprising the data.

To make file accessing more efficient, we have partially inverted the files on two frequently accessed domains, the BUSER and the year field of the action date.

3.3 The Data Sublanguage ALPHA

Having thus selected a model for the data, our next step was to select a query language. Two families of higher level data sublanguages for a relational data base are based on, alternatively, the relational algebra (derived from the algebra of sets) or the relational calculus [Date (1975), Codd (1971b), Codd (1971c)] (derived from the predicate calculus). Codd (1971c) and Date (1975) have compared these two and found the relational calculus to be superior, particularly for use as a target language for a natural language system. The main reason for their choice of the relational calculus as a target language was that the calculus is non-procedural; i.e., a query in the relational calculus conveys little information about how to proceed in searching the data base. The relational algebra is more procedural which makes the automatic construction of a query by a natural language system somewhat more difficult.

A data sublanguage (DSL) refers to the parts of a query language which are oriented strictly towards data accessing as opposed to computation. The host language (e.g., LISP, SAIL, etc.) is the language in which the DSL is embedded. We have chosen to use the relational calculus sublanguage, DSL Alpha, proposed by Codd (1971b). For our purposes, the DSL will be embedded in LISP, so we have modified the syntax of DSL Alpha (as presented in [Codd (1974)] to be compatible with LISP. The syntax of

Relation	Cardtypes	Description
A	76	Flight data summary
F	12,32	Failed parts
I	17,27,47	Installed parts
M	11,21,31,41	Authorization of maintenance action
O	79	Overall summary
R	16,26,46	Removed parts
U	71	Maintenance data summary

Table 3.1 Relations for the 3-M Data Base

the resulting language is presented in Table 3.2 (note that, since our natural language system is aimed only at retrieval of data, we omitted those portions of DSL Alpha which are relevant to data modification). The notation used is BNF; terminal symbols are indicated by capital letters.

The semantics corresponding to this syntax are, briefly, as follows. The function FIND is the top level function of this implementation. It returns an output relation which is a list of the form (⟨name list⟩ ⟨tuple list⟩).

1. The ⟨name list⟩ is a list of domains in the ⟨tuple list⟩.
2. The ⟨tuple list⟩ is a list of output tuples with each tuple in the form specified by the ⟨target-list⟩ specification. The ⟨tuple list⟩ contains only unique tuples; i.e., all duplicates are eliminated.
3. ⟨quota⟩ is the maximum number of tuples permitted in the ⟨tuple list⟩ (if more occur, the list is truncated after sorting).
4. ⟨range-specs⟩ associates tuple variables with specific relations (a tuple variable is a variable which takes as values the tuples in its associated relation). All tuple variables to be used in the FIND function must be declared in ⟨range specs⟩.
5. ⟨target-list⟩ specifies the ⟨fterm⟩'s which make up the tuples in the ⟨tuple list⟩ above.
6. ⟨term⟩ references a specific domain of the relation for the specified tuple variable; i.e., (X HOWMAL) references the "HOWMAL" domain for the variable X. (X (TOTAL)) means that the user defined function TOTAL will be executed for each value of X which permits the summing of several domains, for example.
7. ⟨bool-expr⟩ is a relational calculus expression which must be true for each tuple in the output list.
8. ⟨sort-expr⟩ specifies how the output relation is to be sorted, the domain on which to sort being specified.

We now give two examples of the use of the query language. First we have the query,

(S3.1) Find the total number of hours of unscheduled maintenance for BUSER 3 during 1972.

```
query_stt ::= (FIND quota (range_specs) (target_list) (bool_expr)
        (sort_expr))
quota ::= integer  ¦ ALL
range_specs ::= (tuple_var rel_name) range_specs  ¦ null
target_list ::= fterm  ¦ fterm target_list
fterm ::= term  ¦ (fcn arglist)
term ::= (tuple_var attribute)
attribute ::= domain  ¦ (user_fcn)
arglist ::= term  ¦ arglist term
bool_expr ::= bool_expr  ¦ (quant tuple_var bool_expr)  ¦
        (log bool_expr bool_expr)  ¦ (NOT bool_expr)  ¦ pexpr
quant ::= ALL ¦ SOME
pexpr ::= (pred jterm jterm)
pred ::= NEQ ¦ EQU ¦ LEQ ¦ GEQ ¦ LT ¦ GT
log ::= AND  ¦ OR
jterm ::= term  ¦ const
fcn ::= SUM ¦ AVG ¦ COUNT ¦ MIN ¦ MAX
sort_expr ::= (attribute seq)  ¦ NIL
seq ::= UP ¦ DOWN
```

Table 3.2 Syntax for DSL Alpha Implementation

This translates into the DSL Alpha expression:

(Q3.1) (FIND ALL ((V 0)) ((SUM (V NORMUNS))) (AND (EQU (V ACTDATE) 72)
(EQU (V BUSER) 3)) NIL).

Here V is declared to be a tuple variable on the 0 relation (which contains cardtype 79, monthly summaries). SUM is a built-in function which, in this example, sums the NORMUNS domain over all values of V which satisfy the logic expression.

The second example is,

(S3.2) Find the date and not operationally ready hours for all maintenances which were performed on the same day as a flight.

Our DSL Alpha expression for this is:

(Q3.2) (FIND ALL ((V1 U) (V2 A)) ((V1 ACTDATE) (V1 NORHRS)) (SOME V2
(EQU (V1 ACTDATE) (V2 ACTDATE))) (NORHRS DOWN))

In this query V1 and V2 are tuple variables on, respectively, relations U (cardtype 71, daily maintenance summaries) and A (cardtype 76, daily flight summaries). In the logic expression (SOME V2 ...), we see that V2 is existentially quantified, so a given value of V1 will satisfy the expression only if there exists a value of V2 such that the ACTDATE domains of the two varibles are equal. The sort expression, (NORHRS DOWN), specifies that the tuples in the output relation will be sorted in descending order on the NORHRS domain.

It should be noted that our syntax contains no explicit restriction of the range of a universally quantified variable (e.g., Y in (ALL Y (EQU (X JCN)) (EQU (Y JCN))) as does the syntax of alpha expressions as presented by Codd in (1971c). We have assumed that the universe for such a variable is determined by the monadic (i.e., containing a single variable) predicates for that variable in the logic expression. For

example, in the expression (ALL V1 (AND (GT (V1 NORHRS) 10) (EQU (V1 JCN) (V2 JCN)))) the universe for V1 would be all members of its associated relation for which the value of the NORHRS domain is greater than 10.

3.4 Optimization

The next important question is how to implement this language. Codd (1971c) proposes an algorithm to reduce the relational calculus to the relational algebra, which, as mentioned above, is more procedurally oriented than the calculus. The operations of the algebra (project, join, divide, etc.) are easily implemented, and so we may search the data base by reducing an expression in the relational calculus to one in the algebra and then executing the latter expression on the data base. For this reduction to be practical, however, it must produce an efficient expression of the query in the algebra; so we come to the problem of optimization.

Since the relational calculus is not a procedural language, it is left up to the query system to discover an efficient strategy for searching the data base. While Codd's reduction algorithm provides a relatively straight-forward way to implement a relational calculus language, it is an inefficient way to proceed. Palermo (1972) has identified two major problems in the reduction algorithm: 1) intermediate storage requirements may be extremely large, and 2) a given tuple may be retrieved more than once from the data base. Palermo proposes modifications to the reduction algorithm which minimize intermediate storage requirements and require that no tuple be retrieved more than once. The algorithm uses statistical information about the data base to dynamically determine the order in which the relations in the data base are to be explored (the phrase "exploring a relation" is used to indicate a process of extracting information related to a specific query).

3.5 The Implementation

We have implemented Codd's DSL Alpha following Palermo's algorithm. A flow chart for the program is shown in Fig. 3.1. The operation of the program will be discussed below in more detail as we follow through the execution of two examples.

Our program contains one major enhancement of Palermo's algorithm in our processing of single variable queries (which we call "simple" queries) which do not require intermediate data structures to be formed. For these queries, we form the required output relation as the data base is searched, rather than building an intermediate object as the data base is searched and then forming the output relation after the search is completed. This method results in substantial savings of time and storage, particularly when a built-in arithmetic function (such as SUM) is being used.

Another feature of this implementation is a capability to save the results of a search for future use. This is important, particularly in a system such as this where a large data base is to be queried interactively. In this situation, interesting results from a query may be expected to elicit another query, and this query is likely to reference the tuples just retrieved. This situation can be recognized in the natural language system when pronoun references are being resolved. To allow search results to be saved, a flag may be set (before the FIND function is called) which will cause all tuples satisfying the logic to be stored in temporary relations. The names of these relations (e.g., U0001) are made available to the user (or the natural language system) who may use them as relation names in subsequent queries.

We shall now describe the basic operation of the program in more detail. This will be done by means of two examples, one which requires a simple search and the other a "complex" search. These examples are essentially the same as those used above in the section on DSL Alpha.

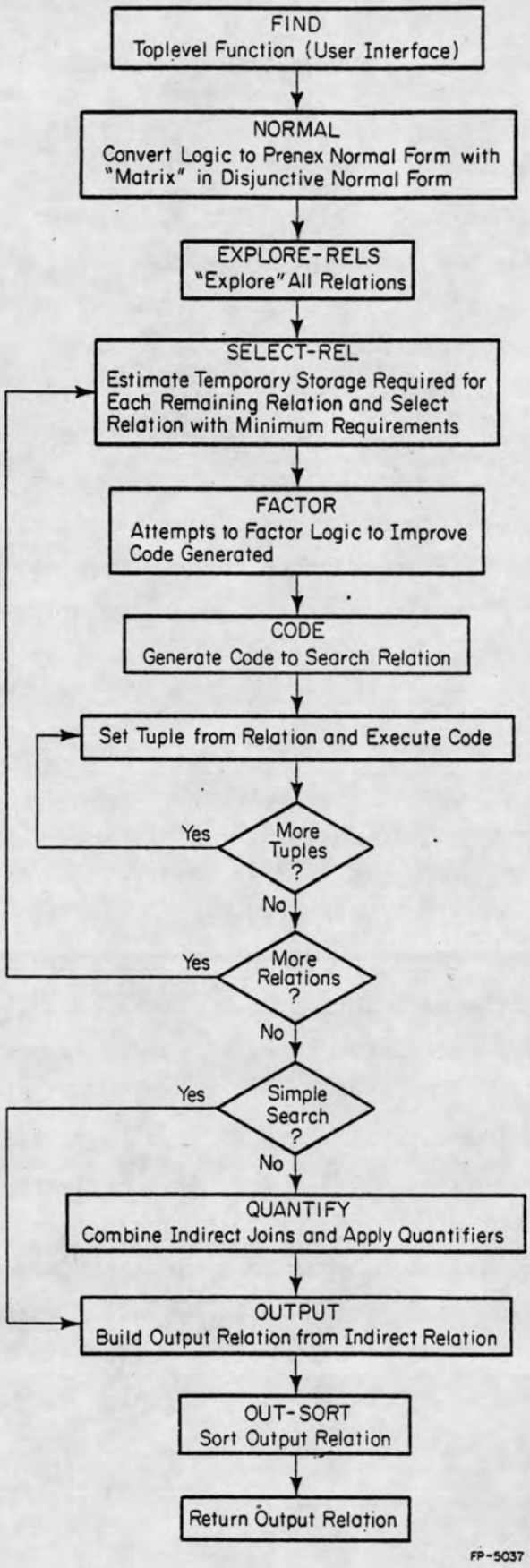


Figure 3.1 Flow chart of relational database system

3.6 Simple Search

Given the query:

(S3.1) Find the total number of hours of unscheduled maintenance for BUSER 3 during 1972 which corresponds to the query language form:

(Q3.2) FIND ALL ((V 0)) ((SUM (V NORMUNS))) (AND (EQU (V ACTDATE) 72) (EQU (V BUSER) 3)) NIL).

The toplevel function FIND first calls the function NORMAL. Since the logic expression is a single conjunct, the expression is returned substantially unchanged. We now enter EXPLORE-RELS in which the actual search is carried out.

Since only one relation, 0, is involved, SELECT-RELS simply returns that relation. FACTOR has no effect since there is only a single disjunct in the logic expression (FACTOR is frequently necessary in other cases since the process of putting a logic expression into disjunctive normal form often causes the same predicate to appear in every disjunct, and evaluation of an expression in this form would obviously be very inefficient). After FACTOR has been exited, the function CODE is entered.

When CODE is called to generate the code for the search, two important things are discovered: 1) It is realized that both ACTDATEYR and BUSER are index domains and some code is generated which will cause only those tuples in relation 0 and with ACTDATE=72 and BUSER=3 to be accessed on disk. 2) It is recognized that this is a simple search, so the code (for the SUM function) is generated to sum the NORMUNS domain for each tuple as it is retrieved. The code generated by 2) is simply:

(SETQ N (PLUS N (GETFIELD NORMUNS))).

Here N is a variable which is initialized to zero at the start of the search and GETFIELD is a function which extracts a given domain (in this case NORMUNS) from a tuple. SETQ is the standard LISP function which assigns the value of its second argument to its first argument (in FORTRAN we might write N = N + GETFIELD(NORMUNS)).

We now exit from CODE and enter a loop where the required tuples are read from disk one at a time with the above expression being executed on each tuple. When the search is completed, we have N = 280.

We save this result and exit from EXPLORE-RELS. We now enter the function OUTPUT where, since sorting is not required, we simply return the output relation:

(((SUM (NORMUNS))) (280)).

3.7 Complex Search

Suppose that we have two temporary relations A0002 and U0001 which have been created by an earlier search and that the values of some selected domains of the tuples in these relations are as shown in Tables 3.3 and 3.4. We wish to determine the results of the query:

(S3.2) Find the date and not operationally ready hours for all maintenances which were performed on the same day as a flight which translates into the query language form:

(Q3.2) (FIND ALL ((A A0002) (U U0001)) ((U ACTDATE) (U NORHRS))
(SOME A (EQU (U ACTDATE) (A ACTDATE))) (NORHRS DOWN)).

This time, NORMAL removes the quantifier (SOME A) from the logic expression and returns it and the remainder of the logic expression (the "matrix", which is, in this case, (EQU (U ACTDATE) (A ACTDATE))). These are passed to EXPLORE-RELS which first calls SELECT-REL which estimates the intermediate storage that would be required by each relation if it were explored first and then selects the relation with the minimum storage requirements. The storage requirements are estimated by following through the code generation process used by CODE; but, instead of generating code, statistical information on the data base (e.g., the number of unique values in a given domain of a relation) is used to estimate the probability that a tuple will satisfy a given predicate. These probabities are then multiplied by the number of tuples to be searched to derive the storage

TUPLE#	ACTDATE
1	3179
2	3180
3	3182

Table 3.3 The temporary relation A0002

TUPLE#	ACTDATE	NORHRS
1	3179	5
2	3179	2
3	3181	6
4	3182	7

Table 3.4 The temporary relation U0001

estimate. In the present case, the only predicate is a "join term" which does not restrict the number of tuples to be retrieved (the reason for this will become more apparent later when we discuss the semijoin), so we see that U0001 will require more storage than A0002. Since we are attempting to minimize storage requirements, SELECT-REL choses A0002 to explore first.

We now enter CODE. Here, again, the predicate is recognized as a join term which is so named because in the translation from relational calculus to relational algebra this predicate translates into a "join" operation on the domains in the predicate. Following Palermo, we have employed "indirect relations" for internal manipulation. In an indirect relation, rather than working with an entire tuple, we use only a "tuple reference number" which uniquely identifies the tuple. In Tables 3.3 and 3.4, the TUPLE# column is the tuple reference number (these numbers are not part of the permanent relations on disk, but are generated as the tuples are input). For example, relation A0002 can be represented as simply the indirect relation (1 2 3). A join is implemented in two stages using indirect relations. First, a "semijoin" is formed for the first relation to be explored. A semijoin consists simply of a list of pairs of the form (⟨domain⟩, ⟨tuple reference number⟩) where ⟨domain⟩ is the domain on which the join is to be performed. In our example, the which which would be generated for this is

(SEMIJOIN address ACTDATE)

where [address] is an address where the results are to be stored. A search of the relation A0002 with this code yields the semijoin:

((3179 1) (3180 2) (3182 3)).

Upon the completion of this semijoin, we return to SELECT-REL where U0001 is found to be the only remaining relation. When CODE is entered, it realizes that A0002 has already been explored and so an "indirect join" is generated:

(IJOIN [address] [semijoin on A] ACTDATE EQU)

where [address] is as above and [semijoin on A] is the semijoin just created. The result of the indirect join is a list of pairs of tuple reference numbers with the numbers in a pair corresponding to the two variables in the predicate and with each pair satisfying the predicate. The indirect relation produced by the above indirect join is:

((A U) ((1 1) (1 2) (3 4)))

where the variable names are carried along to identify the columns of the indirect relation.

In exploring this relation, one complication occurs which we did not mention above. When CODE was called for this relation, it was also discovered that there were two domains of the variable U to be retrieved for the "target list" (i.e., ((U ACTDATE) (U NORHRS))). At that time the code also was generated to retrieve these domains and the corresponding tuple reference numbers, so this information was also retrieved at the same time that the indirect join was being formed.

Having completed the exploration of both relations, we now exit EXPLORE-RELS and enter the function QUANTIFY. In this function, all indirect relations which have been generated in the data base search are combined into one indirect relation and we then apply suitable operators to perform the quantifications indicated in the logic expression. In our example, only one indirect relation was produced in the relation exploration, so we only need to apply the quantification operation.

The existential and universal quantifiers of the relational calculus correspond to the operations of projection and division, respectively, in the relational algebra. In our example, A is existentially quantified so we need to project our indirect relation onto the variable U. To do this, we simply remove the tuple reference numbers for the A0002 relation and eliminate any duplicates in the resulting indirect relation (in this case there are none). This yields the result:

((U) ((1) (2) (4))).

We now leave QUANTIFY with this relation and enter OUTPUT where we assemble the output relation by taking those items in the target list

with tuple reference numbers corresponding to those in the indirect relation and eliminate any duplicate tuples. This gives us the result:

((ACTDATE NORHRS) ((3179 5) (3179 2) (3182 7))).

This is now sorted on the NORHRS domain to yield the final output relation:

((ACTDATE NORHRS) ((3182 7) (3179 5) (3179 2))).

3.8 The Future

Our implementation of the DSL Alpha query language currently provides a relationally complete (as defined by Codd [1971c]) interface between the natural language system and the data base. The system also provides features such as the storage of search results. The basic algorithm on which the implementation is based is reasonably efficient; however, before any large scale use of the software is practical, a number of improvements and evaluations should be made.

First of all, there are several areas of the program where substantial improvements in performance can be affected with some recoding. The area where improvement is most urgently needed is in the use of list structures (which are inferior in terms of both machine time and storage requirements to simpler array structures) for the storage of intermediate results. In addition, it would be desirable if a facility existed for storage of parts of the indirect relations and other temporary data structures on secondary storage (i.e. disk) to allow working with the large temporary relations likely to occur in practical usage of the software (currently, all these objects are stored in core). In the area of evaluation, the method used by SELECT-REL to estimate storage requirements needs to be evaluated to determine if it is adequate, and the overall performance of the system needs to be analyzed to attempt to discover any remaining bottlenecks.

In addition to these considerations, some performance improvements could be achieved through modifications of the algorithm used. Smith and Chang (1975), for example, have described a scheme for the optimization of expressions in the relational algebra, and some of their techniques could certainly be employed in our system.

4. Elements of the System

In this section we explain in detail the operation of several interesting system constituents: (1) the lexicon (an expert on words), (2) the spelling corrector, (3) the translator (which translates a request from paraphrase language--the output of the language understanding system--to data base query language), (4) the output formatter (which automatically selects tabular, graphical or list forms to present output to the user).

4.1 The Lexicon

The lexicon consists of two parts: the DICTIONARY which contains words and certain features associated with them and the DICTIONARY MANAGER which is a package of procedures for accessing, maintaining, and updating the dictionary. The overall organization of the lexicon is shown in figure 4.1.

4.1.1 The Dictionary

Currently, our dictionary consists of approximately 1100 words and phrases with associated syntactic information. A dictionary entry for a word typically consists of the syntactic category that the word belongs to and a list of syntactic features for the word when interpreted under that category. It is a fact of English, as in most other languages, that many words allow multiple senses for a single word. In these cases, multiple entries are made in the dictionary. For example the word CRASH has the following entries:

CRASH N -es

CRASH V -s-ed (intransitive)

This says that CRASH can be interpreted as a noun (N) whose plural is formed by adding the suffix "es" or as an intransitive verb (V) whose inflectional forms can be generated by adding the suffixes "es" or "ed".

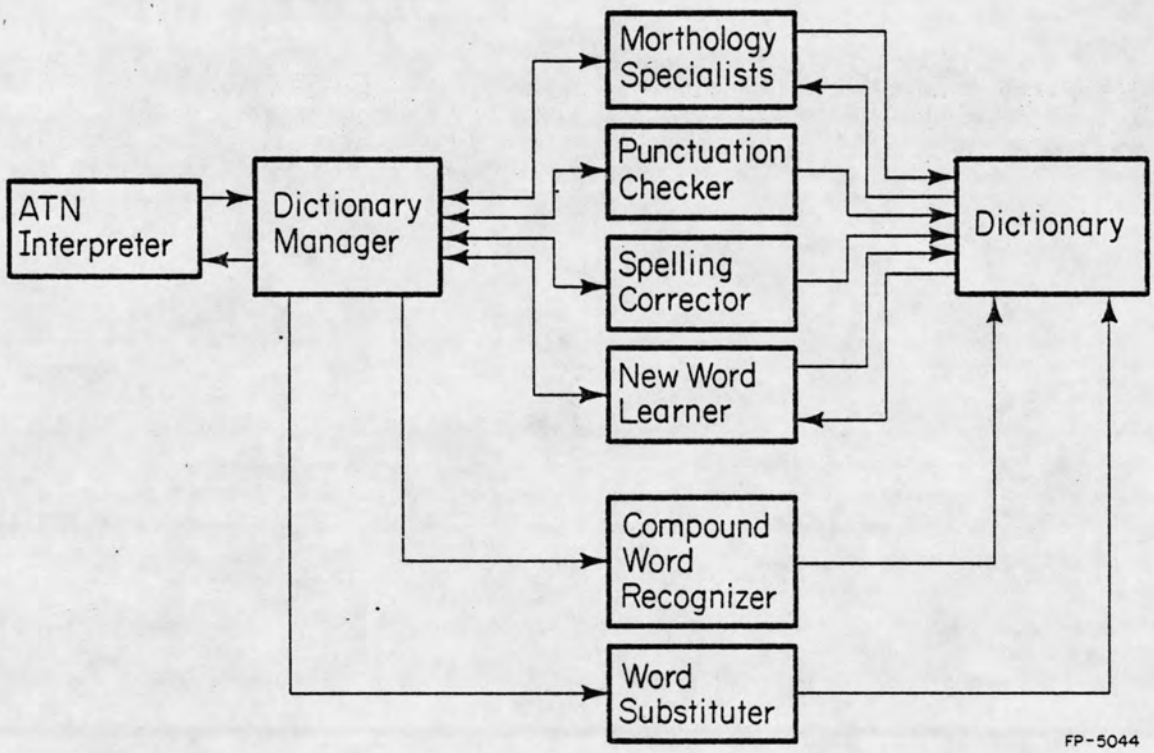


Figure 4.1 Organization of the lexicon

Initially, no entries exist for the regularly inflected forms of words. Such forms are discovered and entries for them generated as they are needed. For example, if the input contains the word CRASHED, a set of programs (morphology specialists) are invoked which discover that CRASHED is the past tense form of the root word CRASH. These programs insert the lexical entry:

CRASHED V (CRASH) (tense past)(intransitive)

The words CRASHES and CRASHING are similarly recognized and their lexical entries generated.

4.1.2 The Dictionary Manager

The dictionary manager is a collection of programs which access, maintain, and update the dictionary. They also act as a filter between the Parser and the users input suggesting alternative words as the 'next word'. As each word of the input is needed by the parser, the dictionary manager checks to see if there is an entry in the dictionary for that word or if the word belongs to one of certain categories which are recognized procedurally, such as the category "numbers". If an entry exists, then the word is passed on to the parser with a list of features associated with the word. These features are those found in the dictionary together with any others suggested by the morphology specialists.

If the word does not have a lexical entry, then a series of morphology specialists are invoked to see if the word is a regularly inflected form of a known word. These specialists use their knowledge of typical English affixes to propose candidate 'roots' for the word. Each candidate is then looked up in the dictionary and, if found, checked to see if it accepts the affix removed.

If this process fails then the punctuation checker is called which examines the word to see if it might contain any embedded punctuation. For example, if the input is TAIL-NUMBER, the punctuation checker would suggest that the two words TAIL and NUMBER be substituted.

If the word remains unrecognized, then the misspelling checker is invoked to see if the word might be a misspelled version of a known word (see section 4.2).

Finally, if the word is still unknown, control is passed to the new word learner. This module interacts with the user and attempts to create a lexical entry for the word.

Two other subsystems generate alternative suggestions for the 'next word': a compound word recognizer and a word substituter. The compound word recognizer is used to map short phrases into single "words". For example, we map the phrases UNITED STATES and UNITED STATES OF AMERICA into the single "word" USA. Just before a new word is passed to the parser, the compound word recognizer checks to see if that word can begin a phrase that it knows. If so, then it checks the rest of the sentence, and if it matches, suggests the alternative 'next word'.

The word substituter provides a similar mechanism - one which can expand a single word in the input into a sequence words. For example, the word DIDN'T is expanded into the sequence DID NOT. The sequence of words substituted can be of zero length, which provides a facility for ignoring words. This facility is used, for example, to ignore extraneous punctuation.

4.2 Spelling Correction

An important part of interactive systems is the flexibility of the user interface: it should be as tolerant of errors as possible. Until workable speech understanding systems are available, the only method for natural language input will be via typewriter keyboard. One part of making this situation as livable as possible is to provide automatic spelling correction at the front end of the system. A fast and effective method for spelling correction has been devised and implemented for the PLANES system.

4.2.1 The Problem

Normally, humans have little problem understanding written sentences which contain even a very large number of errors. The reason for this

is that natural language contains a great deal of redundant information: The most commonly quoted figure is that each letter in an English sentence adds on the average only about 1.7 bits of information [Shannon (1948)]. This low information density is due to the fact that not all letter sequences are English words, and not all words can appear in any position in a grammatical and semantically sensible sentence. While we cannot hope to do as well on really garbled sentences as a human can, if the system can tolerate most typographical errors it should be considerably less frustrating to use.

4.2.2 Solution Methods

Several substantially different approaches to spelling correction can be thought of; and all of the following ones have been tried in existing systems.

(a) Perhaps the simplest approach is to experimentally study what misspellings actually occur in the use of a particular system, and write "filtering programs" that correct those errors specifically. This can be done by simply storing common misspellings as synonyms for the intended word; for systems with a small number of words this is a workable approach.

(b) Another simple approach is to store letters commonly substituted for others (usually based on their proximity on the input keyboard) and try all single letter substitutes, looking for an exact match.

(c) A more time consuming exhaustive search approach is to try all of the "single misspellings" (which appears to cover about 75% of the misspellings which occur in practice). There are four types: missing letter, additional letter, wrong letter, and interchange of adjacent letters. This exhaustive search can be done in a reasonable amount of time only if great care is given to the representation of words in the dictionary, and to the efficiency of the program. It should be noted that the exhaustive approach takes far too long if one wants to correct for all combinations of two "single misspellings"; thus one can not hope to use this method to correctly recognize the word "mispeling".

(d) Yet another approach to spelling correction is based on the idea of pattern matching. The thought here is that enough distinguishing characteristics may remain in the misspelled word to distinguish the correct word from among all the words in the dictionary except the correct one. Many candidates for pattern features have been tried: phonemes, letter pairs, syllabication, length, semantic use in sentence, etc. The advantage of pattern matching is that it has the capability for correcting multiple as well as single errors. A disadvantage is that since the approach is essentially statistical in nature there exists the possibility of mistaken identification, and an inability to correct single errors in a few unusual cases.

The spelling correction algorithm finally implemented for the PLANES system is essentially a LISP implementation of the algorithm used on the PLATO computer aided instruction system [Tenczar and Golden (1972)] with some differences and enhancements. This algorithm is in the pattern matching class.

4.2.3 The Spelling Correction Algorithm

As currently implemented, spelling correction is done a single word at a time, before the input sentence is passed to the parser. Thus the spelling algorithm knows nothing about the context of a particular word in the sentence. In the PLANES system spelling correction must operate under two strong constraints: (1) it must be fast (less than a second per misspelled word) so as not to introduce annoying delays in interaction with the user; and (2) it must do a reasonable job of distinguishing misspelled words from unknown words, since it is anticipated that the dictionary of known words will constantly be added to interactively. The time constraint has been met, and in the current system correctness is insured by asking the user to verify that the algorithm's guess is correct before the sentence is passed to the parser.

The algorithm uses two data arrays: the first contains all the words in the dictionary, and the second contains (in the same order) encoded patterns which are used as the basis of comparison between the dictionary words and candidate misspellings.

The algorithm works by first checking if the word is a member of the dictionary (if so it is correctly spelled); if not it extracts the pattern features of the word and compares these features with the (pre-computed and stored) features of all the dictionary words to see if there are any close matches. The key to the speed of the algorithm is that the pattern comparison can be done very quickly, since the pattern features of each word are mapped on to 2 PDP-10 words (72 bits) in such a way that the exclusive-or of two patterns will give a bit string in which the number of bits that are "1" increases with the "difference" or metric distance between the features of the two words being compared. Thus a numerical measure of the difference between two words can be found simply by exclusive-ording the pattern bit strings and then counting the number of bits that are "1" in the resulting bit string.

For example, one of the pattern features used is the number of letters in the word. The length of the word is encoded into a bit field in such a way that words that differ in length by one letter will give one "conflict bit", those that differ by two letters will give two conflict bits, and so on. Many features of the spelling of a word are used in obtaining a pattern encoding; those features that are considered most important (like length and first letter) are given the longest bit fields so that they have the potential of producing the most conflict bits. The key to the success of the algorithm seems to be that using many simple features in combination gives a description which has greater descriptive power than the simple sum.

Currently, the algorithm uses bit fields for the following characteristics: first character, length, letter content (which letters of the alphabet occur), letter order, consonant-vowel pairs, consonant-consonant pairs, the last letter, doubled letters, and non-alphabetic letters (e.g. dash). The algorithm has been written so that it can be used as loosely or tightly as desired, by adjusting a cutoff limit as to how many conflict bits will be considered as misspellings rather than unknown words. The cutoff limit is adjusted for the length of the word, since there is less information available in a short word than a long one. In actual use

and testing the algorithm's performance is remarkably good: with a 5000 word test dictionary most misspellings of a 10 letter word with as many as three letters wrong or missing came back with only one candidate, which was the correct one. With the actual dictionary of the PLANES system loaded the spelling correction algorithm found the single correct choice for all simple one letter changes to the word "airplane", in addition to the following misspellings: irplaine, arplain, airlpne, airrpplanee, irplan, aiirplne.

4.2.4 Future Improvements

Currently, the algorithm does not handle the typographical error of leaving out a space, which results in two or more words being considered as one: "thelastplane". The simple technique of breaking an unknown word between every letter pair and trying to recognize the resulting fragments should handle most of these cases. Some care is needed in doing this to avoid being confused by letter sequences which are both prefixes and words. For example, if the dictionary contains both the words "air" and "plane" but not the word "airplane", when the user types "airplane" for the first time the system must allow for the possibility that the word is unknown and correctly spelled, and not "outsmart" itself.

A more ambitious future project is to improve the interface between the spelling correction module and the parser so that the syntactic and contextual knowledge of the parser can be used to disambiguate possible misspellings. Then, if the spelling corrector comes up with two candidates for a misspelled word, but only one of them gives a sentence which parses correctly, the system could simply respond: "Assuming xxx is a misspelling of yyy the answer is ...", rather than requiring user verification of every correction.

4.3 The Translator

The translator converts the paraphrase described in the previous section into the database query language. The query language requires

(1) a list of predicates to direct the search, (2) a list of fields to return, and (3) a sort field. The predicates are generated by the translator from the subject, object, and time period in the paraphrase. These predicates include individual plane "bureau serial" numbers(BUSER), plane series codes(TEC), type maintenance(TM) codes, action taken(AT) codes, system work unit(WUCSYS) codes, how malfunctioned(HOWMAL) codes, and actdate(ACTDATEDAY, ACTDATEMON, ACTDATEYR) codes. The fields are discovered by examining the operator(OPER) and action(ACT) positions of the paraphrase. As an example of a translation of a paraphrase, consider:

"Where was the A7 with buser 122 repaired on Feb. 7, 1969?"

Paraphrase:

```
(WHERE (plot NIL)
       (plane (pronoun NIL) (type A7) (buser 122)
              (plneg NIL) (pldam NIL)
             (plmai NIL))
       NIL
       (neg NIL)
       (REPAIRED NIL NIL)
       (time (date (month (2. 31. 31.))
       (day 7) (year 69.))
            NIL)
       (NIL NIL))
```

The planetype list causes the generation of two predicates:

(EQU (W TEC) 'AAFF) and (EQU (W BUSER) 122.).

The time list results in the predicate

(EQU (W ACTDATE) 9038.).

The operator WHERE and the action REPAIRED return the basic frame

```
(FIND ALL ((W M)) ((W ACTDATE) (W ACTWC))
          <place for predicates>
          (ACTDATE DOWN)).
```

Putting the predicates into the frame gives the final result:

```
(FIND 'ALL '((W M)) '((W ACTDATE) (W ACTWC))
      '(AND (EQU (W TEC) 'AAFF) (EQU (W BUSER) 122.)
            (EQU (W ACTDATE) 9038.))
      '(ACTDATE DOWN)).
```

The only difficult task required of the translator is to decide which card types must be searched in the data base. This is easily accomplished if all the fields to be keyed on are unique to a specific card type. However, if the fields could be found on two or more different card types, the Translator must decide which one to use. This turns out to be relatively easy to accomplish by examining the time period context register to see whether the dates are specified according to month-year or month-day-year. This uniquely determines which card type contains all the necessary fields and the correct date specification.

As an example, in the sentences

(1) "How many flight hours did plane 3 log in Jan. 71?" and

(2) "How many flight hours did plane 3 log on Jan. 5, 1971?"

both sentences generate exactly the same paraphrase except for the time period field, namely:

```
(COUNT (plot NIL)
       TIME
       (FLY
           (plane (pronoun NIL) (type PLANE) (BUSER 3)
                  (plneg NIL) (pldam NIL) (plmai NIL))
           Nil)
       (time <date> NIL)      (NIL NIL))
```

where for (1): <date>=(date (month (1. 0. 0.))
(day NIL) (year 71.))

and for (2): <date>=(date (month (1. 0. 0.))
(day 5.) (year 71.))

The translator has the choice of searching two different card types--monthly flight summary cards or daily flight cards. Upon examination of the dates it realizes that to efficiently answer (1) it should search the monthly flight summary cards and to answer (2) it should search the daily flight cards. Hence the following two queries result for (1) and (2) respectively:

```
(1) (FIND 'ALL '((V 0))
                '((SUM (V TOTHRS)))
                '(AND (EQU (V BUSER) 3.))
                      (EQU (V ACTDATEMON) 1.)
                      (EQU (V ACTDATEYR) 71.))
                NIL)

(2) (FIND 'ALL '((X A))
                '((SUM (X FLTHRS)))
                '(AND (EQU (X BUSER) 3.)
                      (EQU (X ACTDATE) 1005.))
                NIL)
```

4.4 Help Files

In any system where information is transferred, some users will need help with procedural problems whether that system resides in a computer or in a filing cabinet. There must exist a reference manual of some sort for the user to quickly find an answer to these problems. If the system is in a computer, the reference manual can be placed in the computer, too. Then it is called a "help file". It can even be made interactive. The advantages of this should be obvious: (1) the user can direct the search for information so as not to be deluged with an entire book, and (2) he or she doesn't have to get up from the terminal and search through a bookshelf for the answer to the problem.

In the PLANES system, help files are disguised to look just like the rest of the system to the user. That is, there is neither a special language to use nor a special mode to enter when the user needs help. PLANES recognizes some questions as referring not to the data base but to its own operation or conventions instead. For instance: "What types of planes do you know about?", or "Can you tell me the abbreviation for 'work unit code'?" These questions are answered using the same machinery as "Which planes crashed in May?" uses. The essential difference is that PLANES looks in different places for their respective answers.

These are the things a user can now request (and receive) help on: (1) abbreviations; the system can give the standard abbreviations for a number of common phrases and vice versa, (2) codetypes; it knows the official names of plane types, maintenance types, etc., and (3) concepts; a few key concepts have explanatory paragraphs available.

5. BROWSER

The specific purpose of the BROWSER System is to provide a tool for measuring, isolating and correcting problems in the maintenance of aircraft by the U. S. Navy. This paper discusses the design of BROWSER. The reader should understand that only a small portion of the design is currently implemented and as the coding process continues the design will probably change. The design process is iterative with each approximation drawing closer to an acceptable solution to the design problem.

BROWSER can be thought of as a complex information retrieval system. The user may enter English language questions into the system and wait for the information to be returned. On the basis of the returned information he will typically ask other queries and eventually converge on the solution to his main question. Alternatively the user may in vague terms describe the area in which he suspects a problem. BROWSER uses this description and "asks" a series of its own questions in an attempt to converge on a problem solution. In either case, the same underlying data retrieval mechanisms are used.

BROWSER was designed to parallel the techniques a human would use to isolate a problem and its causes. The procedure which it uses to process a problem is a coded version of the process through which a human would achieve the same result. The difference, however, is that the human can handle a wide variation in results while BROWSER can handle only those results which can be foreseen, at least in pattern form.

BROWSER tries to strike a compromise. Many problems fall into categories where creative effort is not needed and unexpected results are rare. A well-defined technique for solving the problem exists. Other problems are not understood enough to provide such a technique. BROWSER can handle routine problems almost unassisted while its interactive ability gives the user access to all of BROWSER's facilities for creative problem solving.

5.1 Modes of Operation

The basic action of BROWSER is the isolation of maintenance problems. In interactive mode, the user can supply a series of questions (i.e. a procedure) to isolate the maintenance problem. In the other two modes,

these procedures are pre-stored and applied to collections of data to isolate maintenance problems. Multi-mode design ensures the effective use of 'live' and 'canned' procedures. Little can be said about the procedure a human employs to isolate problems because of human variability. This paper is concerned with the application of 'canned' procedures. Note, however, that 'canned' procedures are typically the coded version of particularly successful 'live' procedures.

The system design incorporates three modes of operation. The Supervisor controls which mode is currently active and will be described later.

Interactive mode takes precedence over the foreground and background modes. BROWSER is communicating directly with the user through the PLANES system. The user is in direct control of all of BROWSER's facilities. Here the emphasis is on fast response to queries. In this mode the user can:

1) get information concerning the use and limitiations of the BROWSER System; e.g. "can you perform standard deviations?"
2) get immediate answers to questions which require only information retrieval; e.g. "who manufacturers part JR1473B?"
3) retrieve answers to previously processed questions;
4) enter a query which requires extensive processing; e.g. "why is turnaround poor at VF-137?"
5) direct various system functions; e.g. "graph the output of the last query."
6) enter data into one of BROWSER's files; e.g. "note that part JR1473B has a verified manufacturer's defect which results in turbine case ruptures."

The second mode of operation is foreground mode which takes precedence over background mode. When the user enters a question into the system and he does not want to wait for the answer or the projected turn-around time is long, the question is entered into a queue. In foreground mode the system uses stored procedures to process queries in this queue. In this mode BROWSER operates with a priority, the same as an accounting program or any other program in a time-sharing computer system.

The third mode is background mode. When there is no work in the

queue and there is no user at a terminal BROWSER enters background mode. It gets processor time only when no other job in the computer system wants it. BROWSER guides itself through the data base searching for maintenance problems. Problems are defined in BROWSER as particular patterns in the maintenance data. In this mode, BROWSER searches for these patterns throughout the data base.

5.2 The System Configuration

Figure 5.1 is a schematic representation of BROWSER. This configuration represents conceptual and functional divisions rather than divisions between chunks of code. Each component will be described separately. The components described in this section will not be described in further detail.

Although the user is central to the total system design, very little of this influence will be elaborated. The PLANES system is responsible for man-machine interaction. It is a complex system and will not be described since it is documented elsewhere.

It is adequate to say that PLANES is a system which understands a subset of the English language and can translate this input into a machine-understandable paraphase. The PLANES system has the ability to do simple data retrieval. For queries in which the user can specify all the information needed to locate the data of interest, the PLANES system can do the retrieval. For queries which involve the application of procedures to isolate the data of interest, the BROWSER system must be activated.

The Input/Output Handler is a collection of routines which controls data transfers. It performs such tasks as locating, loading and positioning magnetic tapes. For instance, if the maintenance data for an A-7 with tail number 123456 during June 1971 is needed, the I/O handler finds the tape and the location of the data on the tape by consulting a directory. It then transfers this data into a temporary disk file so that it can be processed. How the I/O handler does these things is not relevant to this paper.

The Data Base Search Routines are documented elsewhere. Some sample search queries will be presented with an explanation. This is to give the reader a feeling for the low level routines which BROWSER uses.

A description of the remaining components follow. Please note

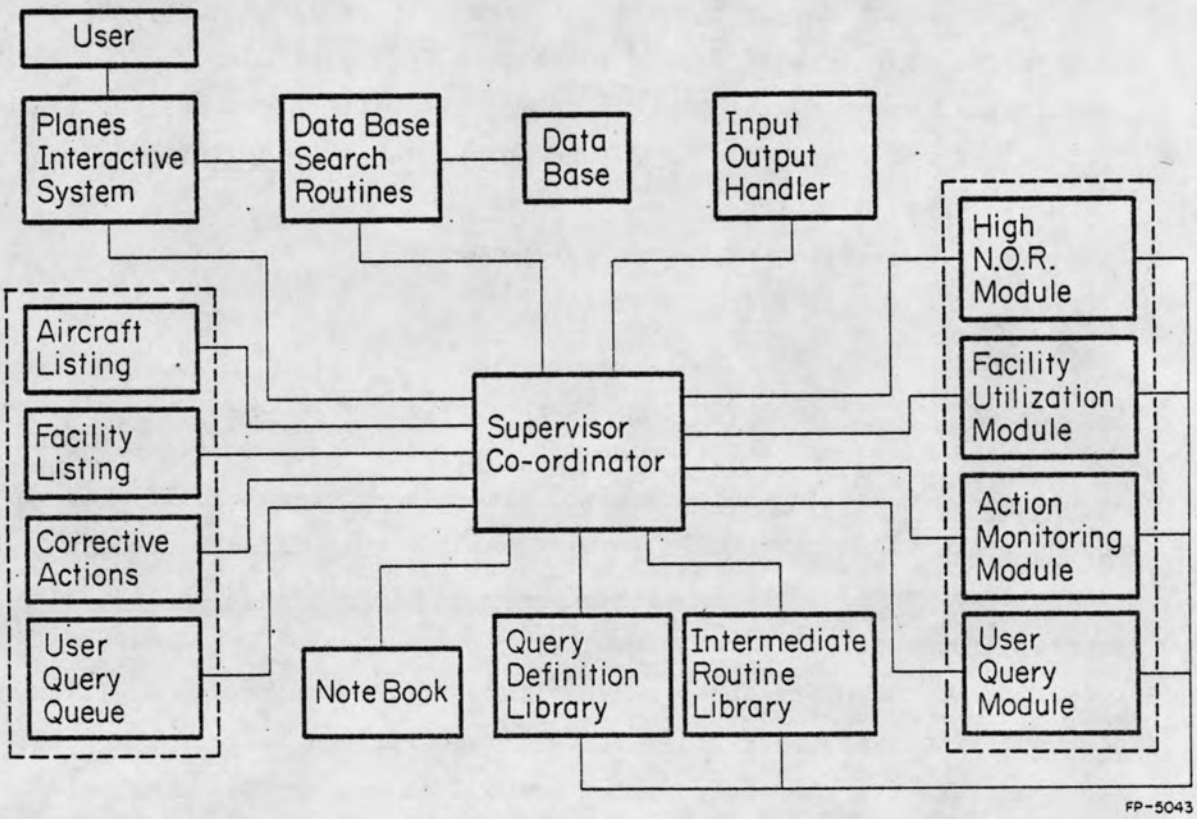


Figure 5.1 Schematic representation of BROWSER

that the terms 'query', 'question' and 'problem' are used interchangeably to improve readability.

5.2.1 The Supervisor

The supervisor is the communication center of BROWSER. Virtually all messages sent between system components must pass through it. After examining the message, the supervisor may alter it before passing it to its destination. There are five functions which the supervisor performs:

1) Data transfers,
2) The setting of system parameters,
3) Temporary answer storage,
4) Mode selection, and
5) Information checking.

Data transfers involve the modification of system files. When a user enters a question which requires extensive processing, it must be stored in the User Query Queue. To accomplish this, PLANES sends a message (the query) to the supervisor along with its destination and purpose. The supervisor verifies that all information for successful processing of the query has been supplied. After making the addition to the queue, the supervisor updates its directories to reflect the change. Data transfers for the other system components are handled similarly.

The supervisor is also responsible for setting certain system parameters. When the user wants a printed copy of the interactive session or a detailed trace of a query's processing or he wants the results of his last question graphed, then the supervisor will set parameters (flags). These parameters tell the supervisor how to go about its business.

Most query processing will involve a substantial turn-around time. BROWSER will work on these queries after the user has left the terminal and will probably finish at a time when the user is not at a terminal. It is the supervisor's responsibility to store the query, the results and other relevant information. The user can then resume his terminal session whenever it is convenient and examine these results.

It is the supervisor which determines which mode (interactive, foreground, background) the system is operating in. The mode depends on

whether there is a user at a terminal and whether there are entries in the User Query Queue. When the mode of operation changes, the supervisor must make sure that no information or processing is lost. BROWSER should be able to re-enter any mode that was interrupted and resume processing as if the interruption had not occurred. This involves saving information on the status of the system when it was interrupted. For example, nearly all queues require the specification of a specific time period. If the processing of a query in foreground mode is interrupted (i.e. a user has initiated an interactive session) one of the items saved is the time period. When the user leaves, the time period is restored and processing continues.

One of the most interesting and useful features of the supervisor is information checking. The supervisor pays special attention to those messages which are data base search queries. Suppose one of the modules wants to know the turn around at VF-137 during June. Suppose also that this information is available directly because a previous query computed it (query results are saved for an indefinite period in the system Notebook). Before passing the data base search query to the appropriate routines, the supervisor checks to see if this information is available in the notebook. This is done by comparing the current query against the queries in the Notebook. A match indicates that the information is already available. In many cases the supervisor can thus speed up the time required to process a request.

5.2.2 Query Management

The management of query processing in BROWSER has three divisions: 1) query sequencing, 2) process supervision and 3) answer refinement.

Query sequencing is selecting the order in which queries will be processed. In a multi-mode system this sequence is determined by the system mode as well as an order within each mode. Sequencing is therefore a function of the User, the Supervisor and the Modules.

The User has the option of setting processing priorities in many system components. This is a way of specifying an order within a system component. In the Aircraft Listing he can specify that A-7s be processed before F-14s. In the High NOR Module he can specify that checks be made for missing inspections before checks for high cannibalization rates. A

useful feature of this priority scheme is that a priority of zero effectively deletes a step. This temporary deletion is useful for skipping over steps which are not relevant to the current query.

The supervisor is responsible for activating queries. The current operating mode determines which query is activated. If the system enters or is currently in foreground mode, the supervisor must select a query from the User Query Queue. If the queue is empty, the supervisor must change modes. Background mode implies that one of the modules will be activated. Typically the particular module and query within the module will be the one following the last query processed. In interactive mode the system is working on queries directly supplied by the user.

(1) The four modules can be thought of as internalized users. For example, the High NOR Module is a series of questions a user might ask to discover why some piece of equipment is unreliable. Each question checks some different aspect of the problem. The general strategy is test and branch. The system asks a question, retrieves data to answer that question and then from an inspection of the data decides whether this aspect of the overall problem is worth processing in greater detail. A negative judgement means that another aspect of the problem should be checked.

Suppose the High NOR Module has processed a query which determines if PlaneX has missed any schedules inspections last month. If the answer is "no" then typically further processing on inspections may be bypassed. If plane X has missed inspections, the module must decide whether this is significant information. If the missed inspections are deemed significant, the module will initiate other queries to determine if similar planes had similar patterns of missed inspections. Further processing may determine that this inspection pattern occurred only on planes that crashed. In this case, plane X would be tagged as having something in common with the crashed aircraft and after further processing and accumulation of results, the user would be alerted to this fact.

(2) The second point under query management is supervision of query processing. This is primarily done by the modules with assistance from the supervisor. Once a query is selected, the active module (e.g. User Query Module) retrieves the query definition from the library. This

definition specifies all the routines necessary to answer the queries. In turn, each routine specifies what information is required before successful data retrieval or other processing can take place (e.g. the time period must be given before data can be retrieved). This process will be detailed later. Presently it is of interest to say that the active module checks to be sure that all prerequisite information is present. If it is not, the supervisor is notified. The supervisor must determine if it can write a data base search query to supply the missing information. If it cannot, the present query is shelved until the user can be contacted and the information supplies. Missing information such as a listing of aircraft which use part X can be retrieved from the data base. However, if the user did not specify the depot at which he wanted to examine turnaround time, the system must wait until the user is available to specify the depot.

When the module decides it has satisfied all the prerequisites, it begins actual processing of the query. Guided by the query definition and intermediate routines it begins to issue data base search commands. Upon completion of each data base search, the module first handles any error conditions which arise. Next it decides whether to pursue this query further and if so in which direction and to what depth. In the user query module the completion of a query signals the end of processing and control goes back to the supervisor. In the other three modules a check is made to see if other queries remain to be processed. If all queries are processed control is returned to the supervisor.

(3) The third point under query management is clarification and refinement of query answers. Much of this topic was presented under the previous two points. When a module discovers that part X had maintenance problems in several crashed aircraft does it stop and announce this fact, or does it alert the user to all current aircraft with similar histories, or does it check the maintenance record of other parts that the manufacturer of part X supplies? This is a fuzzy area and no simple solution is possible. BROWSER allows the user to set the maximum processing depth for queries. Once this depth is reached BROWSER stops. This is not the complete answer since extensive processing will waste time on trivial queries while shallow processing will

miss some problems altogether. The best compromise is to have the user enter into an interactive dialog with BROWSER after it has located some problem at the specified processing depth. The user then uses BROWSER as a tool to decide if the problem is worth investigating extensively. A sample of how this dialog might proceed is given in Appendix B.

5.2.3 The Modules

A fair amount of space has been devoted to the description of the actual functioning of the four modules without describing their organization or usage. These will now be described and hopefully some of the mist will dissipate.

Before designing BROWSER, I became acquainted with the queries that the Navy actually receives. Some of these queries are presented in Appendix F. Logical and processing similarities seemed to categorize most of the queries into one of three groups. The first was the isolation of aircraft and equipment with particularly bad mechanical properties. Here the emphasis was on tracing and eliminating the causes of malfunctions. The high not operationally ready (NOR) module is concerned with these queries. The second category emphasized the efficient use of personnel and repair facilities. The facility utilization module is concerned with these queries. The final category emphasized the monitoring of corrective actions. Here the queries asked if engineering proposals or other corrective actions were actually improving the situation they were designed for. The action monitoring module is concerned with queries in this category.

A fourth module was necessary to provide a method whereby the user could have any query describable by the system answered. The user query module takes an arbitrary query and processes it. If the query is such that one of the other three modules can process it, the user query module merely passes it on to the appropriate module. Otherwise the processing is done locally. This means that the user query module must be more sophisticated then the other's because it must recognize properties of the query to guide its own processing while the others merely accept a query and apply 'canned' routines on it.

A major difference exists between the user query module and the

others. Typically the user module is working on one query while the others have a string of queries which are applied to a subset of aircraft, facilities or corrective actions. A typical query for the user module would be "Does A-7 tail number = 123456 have a particularly bad maintenance record?" Typically the high NOR module would be given a series of aircraft (e.g. A-7's) and be expected to find general maintenance problems or wholesale degration rates.

The queries that the high NOR, facility utilization and action monitoring modules cycle through will now be given. The descriptions are human descriptions of the queries. A few machine understandable queries will be presented later. Some of the queries overlap, many use the same intermediate routines and all use the same set of data base search routines. None of the module queries are unchangeable. Additions, modifications and deletions are expected.

Presently the high NOR module contains the following queries:

A. Checking for Wholesale Degradation of Aircraft
   1) Does a trend analysis of failure and maintenance rates differ significantly from the corresponding rates of new aircraft?
   2) What is the rate of change of failures and maintenances?
   3) Does failure or maintenance rates by work unit code change uniformly? (i.e. Are some systems wearing at different rates?)
   4) Identify aircraft with low/high acceleration of failure/maintenance rates (i.e. is this a good or poor failure/maintenance history?)
   5) How soon will maintenance costs force replacement of this aircraft series?

B. Individual Aircraft/Component Degradation
   1) Cull out the highest failure/maintenance rates by work unit code.
   2) Isolate problem to specified level (e.g. part, unit or subsystem).
   3) Compare against same units in all other aircraft with this configuration.
   4) Compare against same units in this series or type of aircraft.
   5) Compare maintenance histories of crashed and current aircraft which use this part.
   6) Compare inspection schedule with master schedule to determine missed inspections.
   7) Compare inspection histories of crashed and current planes that used this part.

8) Determine if aircraft had all applicable technical directives installed.
9) Compare technical directive installations on crashed and current aircraft.
10) Did crashed and current aircraft use the same maintenance facilities?
11) Were malfunctions due to improper handling of this aircraft?
12) Was there a high rate of cannibalization for the aircraft?
13) Were there any other user specified conditions present?

The facility utilization module presently contains the following queries:

1) Compare the cost of repairing or reworking parts in maintenance facilities to the cost of replacement.
2) Compare the failure/maintenance rates of reworked/repaired assemblies with the rates of new assemblies.
3) Compare the average turnaround time by work unit code across maintenance facilities.
4) Find maintenance facilities with high NOR aircraft then:
   a) Does a high no defect or no repair rate exist?
   b) Find the percent of turnaround due to awaiting parts.
   c) Find the number of aircraft serviced or cannibilized by this facility which had no flight hours in the last 30/60/90 days.

The action monitoring module presently contains the following queries:

1) Compare failure/maintenance rates before and after a corrective action was performed by series.
2) Compare failure/maintenance rates before and after a corrective action was performed across all aircraft incorporating the action.
3) Were the post corrective action failure/maintenance rates significantly better than those predicted by trend analysis without the corrective action?

5.2.4 The Domains

Given the apparatus for accepting and processing queries, how does

the supervisor know what to do and when it is finished? The domain of discourse must be specified in some manner. Given the four modules just discussed, a fairly obvious way is to list all the elements of the domain associated with each module. Every element of every domain has a processing priority associated with it. In the absence of any other information the supervisor simply scans down the domain and selects the entry with the highest priority.

The high NOR module requires a domain consisting of all plane types known to the system. This is a list of all the valid type of equipment codes (e.g. TBCA is the code for A-7A series aircraft).

The facility utilization module requires a domain consisting of all operational maintenance centers. This si a list of all organization codes which denote maintenance centers (e.g. T71 is the AIMD at Pensacola).

The action monitoring module requires a description of all the corrective actions which are to be monitored. Each action is represented by a machine understandable description of the problem, a description of the action and the goal that the action is to achieve. An example of this will be presented later.

The user query module requires only a queue of the queries that are currently in the system waiting to be processed. An example of this type of query will be given later.

In all four cases it is the supervisor which decides which module is to be activated and selects an element of the appropriate domain for processing.

5.2.5 The Notebook

At various times I have referred to the by-passing of certain routines if the goal of such routines is already known. If the number of A-7s is already known, why run the code which will count A-7's in the data base? The mechanism through which this is accomplished is the notebook. The notebook is divided into two sections. The first contains processed queries and the location in which results are stored. The second contains facts and relations which have been shown to exist.

The first section of the notebook can itself be broken into two

sections depending on whether the result of the query is a single number response or a collection of data which satisfies the query. In the latter case a temporary file is created to contain the data while in the former the number is stored in the notebook with the query. "How many A-7s are in the database" would return a number X which is stored in the notebook. "How many A-7s had bird strike damage?" would return a collection of data records, each of which would give information on an A-7 that suffered bird strike damage. These data records would be stored in a temporary file and the name of that temporary file would be saved with the query.

Before any query is processed the Supervisor compares the query against these stored in the notebook. A match means the query was already issued and the results are immediately available. The utility of this method is shown by the interactive dialog of Appendix B. Here the user is continuously referring to the results of the previous query. (e.g. "print all of them but show me..."). If results were not saved, the previous query would have to be processed again before its result could be used by the present query. Note that I have presented a simplified version of this section of the notebook. Each query may actually contain several "sub-queries" and several temporary files to reach a result. For example "how many A-7s suffered bird strike damage and had over 100 hours NOR?" would require a file containing A-7s with bird strike damage and a file containing A-7s with more than 100 hours NOR and the query result is a file containing the intersection of the previous two files. Also note that file management is important because temporary files will accumulate rapidly. BROWSER will let the user specify an expiration date after which these temporary files associated with a query can be deleted. In addition, BROWSER keeps frequency of usage data on each file and will save frequently used files especially if it took a lot of processing time to generate.

The second section of the notebook contains facts and relationships which are known to exist. The user is responsible for entering this data and may do so whenever the system is in interactive mode. Here you will find information on parts which are known to be bad and the effects they create or correlations between situations. Examples of these entries will be given later. The Supervisor watches the performance of the system.

If the system tries to prove a fact or relationship that is already known, the supervisor interrupts processing, supplies the known data and returns control to the appropriate component. Suppose part X is manufactured with a defect which causes turbine-case-ruptures. If the high NOR module is trying to determine causes of turbine-case-ruptures the supervisor will tell the module about part X. The module will not duplicate effort by re-establishing part X as a bad part. The mechanism here is a simple matching process. Each bad-part entry in BROWSER'S notebook supplies information as to the system, subsystem or unit affected, the type of failure and the part number (among other things). A match in any of these area's means that this part MAY be involved in the problem.

A final point concerning the notebook is its use with hypothetical questions. There are times when you want the answer queries with ficticious or modified data as opposed to real world data. As a simple example, suppose you know part X has manufacturers defects and that this condition will be corrected. You now ask "suppose part X had no defect, what would the mean time between failures be on A-7 aircraft?". BROWSER would tag the appropriate item in its notebook as not being relevant for the present query. Processing would continue as normal except that part X would not contribute to the mean time between failures. Each time a failure which involved part X was found the supervisor would cause it to be ignored. The addition, deletion or modification of items in the notebook to suit the needs of a hypothical question would be mostly a matter of tagging appropriate items. This would not interfere with ordinary processing because the tagged items would be applied only to a particular query.

5.2.6 The Libraries

The library structure is such that only one copy of a routine to perform a given function exists. Modifications and updates of routines are therefore automatically reflected throughout the system.

The heart of any module is really the name of a query and the information necessary to direct query processing. The modules contain code to sequence queries and analysis results but the actual query is in the query definition library. The definition of a query contains the name

of the query, parameters which must be supplied for successful processing of the query and a string of intermediate routine names. The intermediate routines are the steps which must be performed to answer the query. The English version of the queries that BROWSER knows about were given in the section on modules.

The routines needed for answering queries are in the intermediate routine library. Each routine performs a complete function and may be used by any number of queries. These are still fairly high level routines which use many data base search commands to complete their functions.

The lowest level routines are the data base search commands. These are the only routines which actually manipulate the data base or perform comparisons on temporary files. Every detail of operation must have been specified by one of the higher level routines for the search commands to work.

Further description the contents of the libraries is given in the "coded examples" section of this paper.

5.2.7 The Data Base

The data base is a term which includes a variety of files. The size of the data base is huge in terms of bits of data stored. This is a major component in estimating how long it will take to process a query. Its size makes it mandatory that searching and processing of the data base be done as efficiently and as seldom as possible. This is one reason that BROWSER saves the results of searches and other data that might be referred to frequently. Files that are in the data base are:

1) The 3-M data base which encompasses all aircraft maintenance, flight and readiness data.
2) The 3M reliability and maintenance reports. (A summary by aircraft series of failure rates, maintenance manhours and flight data. This provides a measurement of the "average" aircraft for comparison purposes.)
3) Aircraft configuration file. (This is a file which specifies which parts are on which plane.)
4) Inspection schedules.

5) Hierarchy of command. (Which squadrons compose which wing, etc.)
6) A parts file. (This is a cross reference for names, numbers, and interchangeable parts.)
7) Manufacturers file. (Identifies parts supplied by a specific manufacturer.)
8) Technical directive file. (A list of all the engineering change proposals which were to correct some maintenance problem on aircraft.)

5.3 Coded Examples

In this section I will endeavor to present samples of what actually gets passed between the supervisor and the other components of the system. For clarity, exact code will not be given but a close approximation to the LISP functions is presented.

Suppose BROWSER has just started processing a user's query. The typed sentence was "Did the A-7 with tail number 123456 miss any inspections during August 1972?". The query was accepted by the PLANES system and stored in the user query module as:

```
((Reference # 1073)   (Processing-Priority      6)
                      (Query  Find-missed-inspections   )
                      (Tail # 123456)
                      (Planetype  A-7)
                      (Time-period Aug. 72    Aug. 72))
```

The PLANES system does an analysis of the sentence and decides (on the basis of keywords and the type of information supplied) that the user's request best matches BROWSER's query which locates missing aircraft inspections. The user supplied information is identified and given the name of an internally recognized variable. The reference number is supplied by PLANES so that this query can be uniquely identified and a default processing priority of 6 was supplied. As luck would have it this is the only user query in the queue and BROWSER begins to work on it at once. The supervisor taps the user query module on the shoulder and announces that it is time to go to work. The user module decides not to pass the query to another module because it deals with one airplane only. The module then

retrieves the definition of the find-missed-inspections query from the appropriate library. After filling in the blanks, this query looks like:

```
(Find-missed inspections
((Tail #123456) (Planetype A-7) (Time-period  Aug. 72  Aug. 72))
(Get-inspection-schedule     Result 1)
(Get-actual-inspections      Result 2)
(Compare    (Result1    Result 2) · (Return  Result 3)))
```

In English this reads: to find missed inspections for A-7 tail #123456 in August 1973 1) find the schedule of inspections that should have been performed on this plane at that time, store it in a temporary file and put the name of that file in the variable "Result1". 2) Find the actual inspections performed, store the results and put the file name in the variable "Result 2". 3) Compare the two files and store the places where they differ in a file whose name is in the variable "Result3". 4) Return the name of the file in "Result3" to the user query module. The user module will pass the name to the supervisor who gives the name to the PLANES system. The user is then shown the contents of the file. Note that if the user had only supplied the tail # but the planetype was also necessary, the user query module would have complained to the supervisor that it needed more information. The supervisor would issue a data base command which would find the planetype from an obscure list of tail numbers. If the supervisor could not supply the missing information, the user would be required to supply it. Having successfully filled in the query frame, the user query module would retrieve the definitions of the intermediate routines. After filling in the blanks, one looks like:

```
(Get-inspection-schedule
((Tail # 123456) (Planetype  A-7) (Time-period  Aug. 72  Aug. 72))
(Find-tape  Tapename)
(Find  Planetype-inspection-schedule  Tapename Location)
(Output-to-temporary-file location temp-file-name)
(Return  temp-file-name))
```

In English this reads: 1) Find the tape which contains the necessary information. 2) Locate the exact information on the tape and remember its location. 3) Read the information into a temporary file. 4) Return the

name of the temporary file.

Having successfully filled in all the blanks it can find, the user query module passes control to the instantiated query which performs the process just described. Note that the intermediate routine uses data base search routines which look up the needed tape in various indexes, have the tape mounted and do the data transfers.

Suppose that at some later time the user has decided that too many inspections have been missed on the A-7 aircraft. He has issued bulletins and changed policy but wants to know if the situation is improving. Thru the PLANES system he makes an entry in the corrective actions listing which looks like this:

```
((Problem      Find-missed-inspections
      ((Tail #123456) (Planetype A-7) (Time-period Aug. 72  Aug. 72))
      (Missed-Inspection-Data   Results Monthly)
(Goal     (Reduction        10%)))
```

This tells the action monitoring module to keep track of the missed inspections problem. Here the module is told that the find-missed-inspection query was run for the given airplane and the result was unacceptable to the user. That results which were found unacceptable are pointed to by the variable "results". The module is to run the find-missed-inspection query monthly until the inspection record has improved by 10%. It is then to notify the user that his goal is achieved. "Results" also contains the monthly results that the module has received so that the user may see the rate of improvement up to the present.

User created entries in the notebook would look something like:

```
(Bad-Part     (Part           Turbine-Case)
              (Location  (WUC   11340))
              (How-Malfunction       Rupture)
              (Manufacturer       Pratt-and-Whitney))
(Relationship (Correlation     parameters1
              (Find-missed-inspections     parameters2)
              (Failure     (Part     torque-link-strut)
                           (Location  (WUC   11130)))
```

This first entry says that Pratt & Whitney turbine cases have been

identified as a part which has a particularly bad maintenance history. The system of the aircraft which uses this part has the Work Unit Code of 11340. The way the part typically fails is by rupturing. The manufacturer is given so that other products from this manufacturer may be traced if necessary.

The second entry identifies a relationship that has been found to exist. Here there is a correlation between a pattern of missed inspections and the failure of torque-link-struts. The variable "parameters1" points to more specific information concerning the correlation such as what the exact correlation is, when it was found, which routine was used to find the correlation. The next two elements of the notebook entry identify the items between which the correlation exists. To recover the exact pattern of missed inspections you would use the find-missed-inspections query with the parameters which are pointed to by the variable "parameters2". Note that the results of this query may still be available in the temporary file created the first time this query was processed. This entry might look like:

```
(Query-results-saved (Find-missed-inspections
     ((Tail # 123456) (planetype  A-7) (time-period Aug 72  Aug 72)))
     (Temporary-file-name     Q1739)
     (Expiration-date      15Jan76))
```

Any query which matches the specifications in Line 2 need not be processed since the results are already stored in File Q1739 and will be available until at least January 15, 1976.

The final example is one of the data base search commands. A command such as the following could be constructed by the PLANES system directly and need not ever been seen by BROWSER. The input sentence would be something like "Find all action organizations in which tail number 158664 was serviced between January 1970 and June 1970."

The data base search command would look like:

```
(Find all (file maintenance-actions)
    (such-that (equal tail # 158664)
               (time-period  Jan. 70   June 70))
    (return  action-organizations))
```

5.4 Conclusion

The BROWSER system as described has several features which should give considerable aid in isolating and correcting maintenance problems. One of the system's strong points is the use of the PLANES system so that the user has easy access to relevant information without having to code programs himself. The system strives to return only as much information as is needed to answer his queries. A system which answers a query with 100 pages of printout is not performing a service to the user. It is the interactive nature of the system with the user in control that gives this system its power.

APPENDIX A: Augmented Transition Networks

A traditional model for recognizing sentences in certain types of language has been the "transition network". One of these can be drawn on a piece of paper as a group of circles with labeled directed lines connecting them (see figure A1). The circles are called "states". Generally, one will be designated the "start state" and another the "final state". The lines are called "arcs" and represent ways to get from one state to another if the label is matched by the current input word. Every time an arc is taken, the input advances to the next word.

Now, the transition network in Figure A1 can recognize such sentences as "Bob ran." or "Rain falls.", but nothing more complicated. Indeed any transition network is limited to a small subset of possible English sentences. Why is this? One reason is that sentences can have other sentences embedded in them. And those sentences can have sentences embedded in them. And so on. For instance, consider "The man who owned the dog which bit the thief called the police." This sentence has three levels of embedding.

If we allow the transition network to be recursive, this sentence (and others) can be recognized. We allow the arcs to have labels which are state names as well as labels which are parts of speech. Then when a state-labeled arc is to be taken, the name of the state pointed to by the arc is put on a push-down list, and we jump to the state named on the arc without advancing the input. This is similar to a subroutine call and is called "pushing to" the state. If we reach a state with a "/1" in it and none of the arcs out of it can be taken, then we jump to the state named at the top of the push-down list, again without advancing the input. This is called "popping". If, however, the push-down list is empty and we are out of words at the input, we say that we have recognized the sentence.

An example might make this a little clearer. We will use the recursive transition network in Figure A2 to recognize the example sentence given above. Please refer to Table A1 as you read this example.

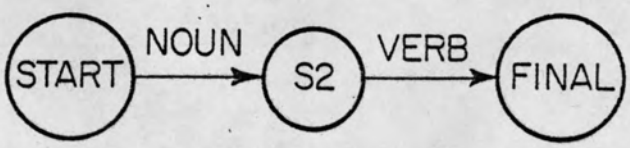
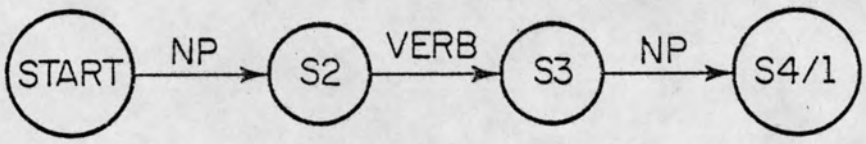
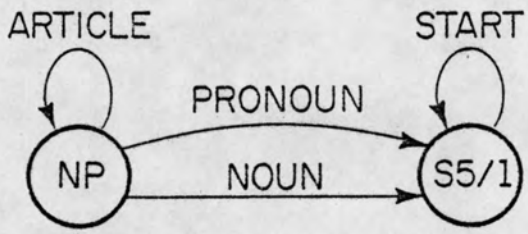


Figure A1 A transition network





FP-5040

Figure A2 A recursive transition network

TABLE A1

STEP	PRESENT STATE	INPUT WORD	PUSH-DOWN LIST (TOP OF LIST TO LEFT)	NEXT ARC TAKEN
1	START	THE	()	PUSH TO NP
1	NP	THE	(S2)	ARTICLE
3	NP	MAN	(S2)	NOUN
4	S5	WHO	(S2)	PUSH TO START
5	START	WHO	(S5 S2)	PUSH TO NP
6	NP	WHO	(S2 S5 S2)	PRONOUN
7	S5	OWNED	(S2 S5 S2)	POP
8	S2	OWNED	(S5 S2)	VERB
9	S3	THE	(S5 S2)	PUSH TO NP
10	NP	THE	(S4 S5 S2)	ARTICLE
11	NP	DOG	(S4 S5 S2)	NOUN
12	S5	WHICH	(S4 S5 S2)	PUSH TO START
13	START	WHICH	(S5 S4 S5 S2)	PUSH TO NP
14	NP	WHICH	(S2 S5 S4 S5 S2)	PRONOUN
15	S5	BIT	(S2 S5 S4 S5 S2)	POP
16	S2	BIT	(S5 S4 S5 S2)	VERB
17	S3	THE	(S5 S4 S5 S2)	PUSH UP
18	NP	THE	(S4 S5 S4 S5 S2)	ARTICLE
19	NP	THIEF	(S4 S5 S4 S5 S2)	NOUN
20	S5	CALLED	(S4 S5 S4 S5 S2)	POP
21	S4	CALLED	(S5 S4 S5 S2)	POP
22	S5	CALLED	(S4 S5 S2)	POP
23	S4	CALLED	(S5 S2)	POP
24	S5	CALLED	(S2)	POP
25	S2	CALLED	()	VERB
26	S3	THE	()	PUSH TO NP
27	NP	THE	(S4)	ARTICLE
28	NP	POLICE	(S4)	NOUN
29	S5	--	(S4)	POP
30	S4	--	()	POP

Starting in state START, the first arc (the only arc in this case) we see is labeled "NP" (for noun phrase). This is the name of a state so we put "S2", the state pointed to by that arc, onto the push-down list. Then we jump to state NP without advancing the input, i.e. we push to NP. The first word in the sentence is "the", an article, so the ARTICLE arc is taken. Notice that we are in state NP again. We look at the next word: "man". This word is a noun, so we take the arc labeled "NOUN" to S5. Now S5 has a "/1" in it, which means that we must pop if we can't take any of the arcs. But we can take an arc, and here is where the recursion comes in.

The arc is labeled "START", the name of the first state we saw. Not to be intimidated, though, S5 goes on the push-down list (which has two states on it now), and we push to state START. We immediately push to NP (S2 goes on the push-down list) and see the word "who". This makes us take the PRONOUN arc to S5.

The next work, "owned", wouldn't be recognized by taking the START arc (try it!), so we pop. That is, we jump without advancing the input to S2, the state on top of the push-down list. Since "owned" is a verb, we can take the VERB arc to S3, where we push to NP again. The push-down list now looks like this: (S4 S5 S2), with the top of the list to the left. We are now at step 10 in Table A1.

"The" and "dog" are recognized by the ARTICLE and NOUN arcs, leaving us in S5. We push to START, then to NP and take the PRONOUN arc for "which". We pop to S2 and take the VERB arc for "bit", then push to NP. From here we recognize "the" and "thief" with the ARTICLE and NOUN arcs (a common construct apparently) and pop to S4. Since there are no arcs out of S4 but there is a "/1", we have to pop again. After three more pops in a row we're in S2 looking at "called" on the input. We take the VERB arc then push to NP and take the ARTICLE and NOUN arcs for "the police". Popping from S5 leaves us in S4 with an empty push-down list. One more pop and (since the push-down list is empty) we've recognized the sentence.

As one can see, recursion tends to make things difficult to follow--and this was a ridiculously simple network. It should be obvious that even this simple network can recognize a certain type of sentence, no matter how many levels of embedding it has.

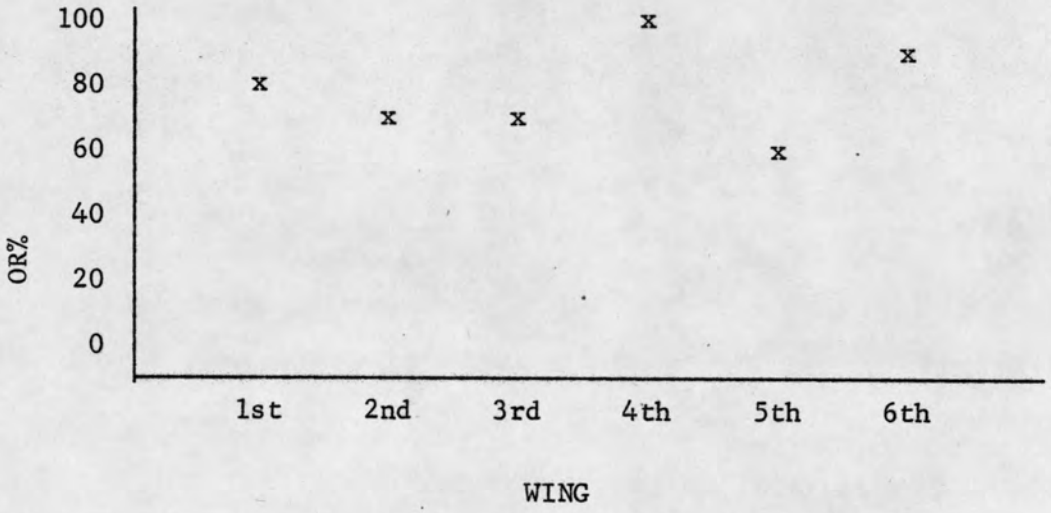
There are many other instances of sentences where processing of part of the sentence must be suspended while the same network processes another part. But since we don't want to just recognize sentences as belonging to a language, we require additional power. We can expand the concept of recursive transition networks.

First, we can put conditions on the arcs which have to be satisfied before the arc can be taken. Second we add structure-building machinery which allows us to save parts of the sentence in registers, to move parts of the sentence around to form its deep-structure, and, in our PLANES system, to form commands to our lower level data base language which correspond to questions asked in English.

This expanded concept is called an "augmented transition network" or ATN. For those of you who are matehmatically inclined, it has the power of a Turing machine (see Minsky (1967), Chapter 6).

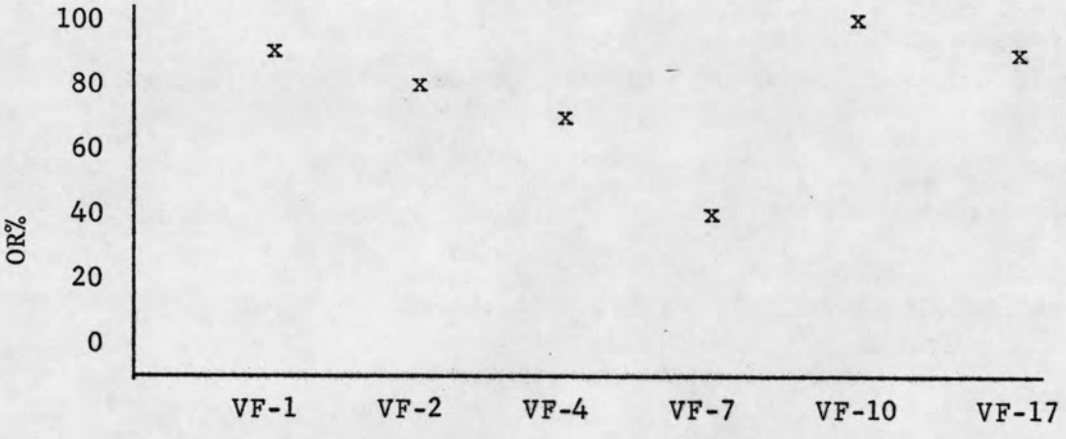
APPENDIX B: Proposed Sample Dialogue for the PLANES System

Show OP RDY% by wing last month



(Would you like to see the exact percentages?)
No

Show OP RDY% for 5th wing by sqdn.



(Would you like to see the exact percentages?)
Yes

Squadron	OP RDY %
VF-1	90
VF-2	82
VF-4	71
VF-7	39
VF-10	94
VF-17	83

Give me hard copy
(output scheduled)

What was NORMS, NORMU and NORS for VF-7?

	%NOR	Tot hrs./month
NORMS	= 17	1304
NORMU	= 19	1768
NORS	= 54	3170

(Did you want to see a detailed breakdown by BUNO?)
No, print it
(output scheduled)

What AIMD is associated with VF-7?
(VF-137)

Show EMT, AWP, MMHRS and TAT for AIMD VF-137.

Total EMT = 2737 hours
Average AWP = 78 hours
Total MMHR = 8417 hours
Average TAT = 91 hours

(Would you like to see all 571 maintenance actions performed at VF 137 last month?)

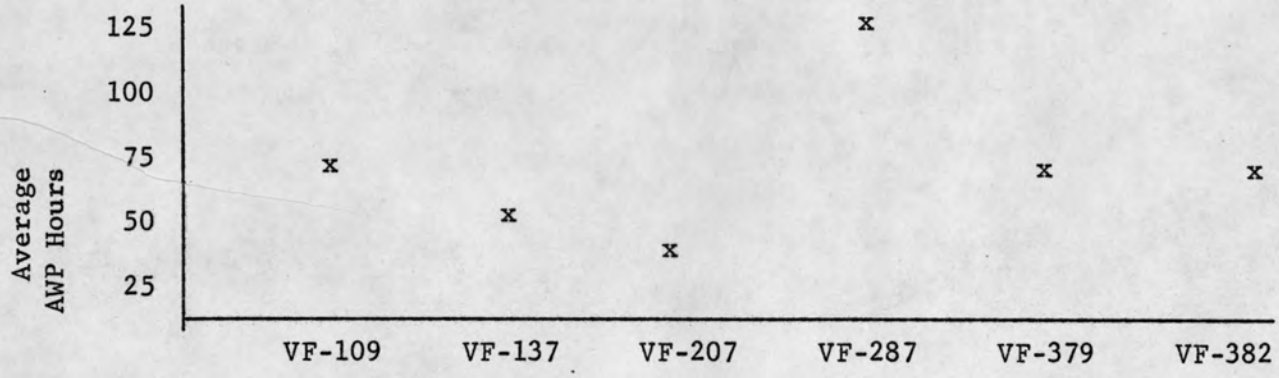
Print them by WUC
(output scheduled)

Show AWP by part for VF-137.
(There are 117 parts. Do you want to see all of them?)

Print all of them but show me the parts with the 5 highest AWP.
(output scheduled)

Part #	NOMENCLATURE	%AWP	AWP Hours
123	Inertial Data Set	57	2036
456	Pneumatic Coupler	11	208
789	Torque Link Fitting	9	159
101	EPP Transformer	8	126
112	Canopy Latch	4	48

Show AWP for Part 123 at Other AIMD's



(Do you want to see the exact AWP hours?)
No, print them

APPENDIX C: Code for the AMOUNT Network (from section 2.1.2.)

```
(defatn AMOUNT
        ; e.g. more than three ==> ((< 3.))
        ; more than three but less than 5 ==> ((> 3)(< 5))
        ; three or fewer times ==> ((< = 3))
        ; four ==> ((= 4.))
        ; any  ==> ((> 0.))
        ("default-registers" (predicates nil) (rel nil) (# nil))
        ((*AMOUNT (wrd (any some)
                       (setr rel '>)
                       (setr # 0.)
                       (to AM:END))
                  (wrd between t (setr rel  < >) (to AM:< >))
                  (cat comp
                       (not (wrd between))
                       (setr rel
                             (selectq *
                                      (atleast '>=)
                                      (atmost '<=)
                                      (lessthan '<)
                                      (greaterthan '>)
                                      (exactly '=)
                                      nil))
                       (to AM:REL))
                  (jump AM:REL t (setr rel '=)))
         (AM:< > (cat integer t (setr # *)  (to AM:< >:1)))
         (AM:< >:1 (wrd and t  (to AM:< >:2)))
         (AM:< >:2 (cat integer t
                        (setr predicates (list  (list '> (min (list $# *)))))
                        (setr rel  <)
                        (setr # (max (list $# *)))
                        (to AM:END)))
         (AM:REL  (cat integer t (setr # *)  (to AM:#)))
         (AM:#  (wrd (time times) t (to AM:#))
                (cat conj (eg $rel '=)  (to AM:CONJ))
                (jump AM:AMT))
         (AM:AMT (cat conj t (to AM:AMT1))  (jump AM:END))
         (AM:AMT1 (push *AMOUNT t
                        (setr predicates (list *))
                        (jump AM:END)))
         (AM:CONJ ; three or fewer ...
                  (wrd (fewer less) t
                       (setr rel '<=)
                       (to AM:END))
                  (wrd more t (setr rel '>=)  (to AM:END)))
         (AM:END (wrd (time times) t (to AM:END))
                 (pop (append (list (buildq (+ +) rel #))
                              $predicates)))))
```

APPENDIX D: The PLANES Data Base

The original data base received from 3-M (in July 1974) was a copy of one month's transactions which passed thru 3-M. However it was impossible to use this for any interesting question answering because the data was not complete. Even questions which spanned a one month interval could not be answered because the one month sample contained corrections for previous months, data for last month and only part of the current month's records. Most importantly however, very few interesting questions could be asked if aircraft histories were not available.

It was decided to obtain a data base from which we could answer interesting questions. After contacting the Naval Air Safety Center in Norfolk, Virginia we obtained the description of 24 crashed aircraft - 12 A-7s and 12 F-4s. A request was then sent to 3-M for the maintenance histories of these 24 aircraft plus 12 aircraft with poor maintenance records plus 12 aircraft with good maintenance records. When the data arrived 3 months later there were several problems. First it was written in a density our equipment could not handle and had to be rewritten by IBM equipment on the University of Illinois campus. The tapes were formatted as print tapes. That is, the records were all 135 characters long with titles and page numbers included. This is fine for printing on a lineprinter but is not easily machine usable. Deletion of blanks resulted in records of 80 characters each. This is a significant saving when one considers that the original tapes would have occupied almost a full disk pack. A serious problem was found during processing, data for 20% of the aircraft were missing. Since the absence was noted on all tapes, it is unlikely it was a machine error. In any case, by the time the error was discovered, it was too late to reorder the tapes (3 months was an unacceptable wait) and we had to carry on with what we had.

Before it was possible to answer questions from the data base it was necessary to understand what was included in the data we had. Almost two years spend in thumbing through various Navy manuals and telephoning personnel at 3-M was necessary to approach a reasonable

comprehension of the data base. I decided to structure our data base around the functions that the records denoted. I wrote an eleven page internal paper which described in detail how the file structure was to be created, what the reformatted records would look like, how to process the tapes and how to identify the aircraft on which we had data [Conrad (1975)].

Each file is organized by 1) record function 2) planetype (e.g. A-7 or F4) 3) tail number and 4) time period. For example, one file would contain all the failed parts found on tail number 156664 (which was an A-7) in 1972.

The complete list of record functions is:

1) Maintenance actions
2) Installations
3) Removals
4) Failed parts
5) Daily flight summary
6) Daily maintenance summary
7) Monthly flight and maintenance summary
8) Summary of failures by work unit code for the series

Appendix E reproduces an internal paper I wrote which describes the data included in each summary plus an example of typical data for F-14 aircraft.

Before designing a question answering system, it is desirable to have an idea of what kind of questions will actually be asked of the system. To this end a study done by NAILSC was consulted [NALDA/NAILSC (1974)]. It listed the data elements that were frequently retrieved from the data base along with crude versions of the questions personnel wanted answered. The study of this report led to three more internal papers.

First I compiled a listing of data elements which occurred in the data base. This included the definition and usage of the field and in some cases the definition of codes which could occupy that field. Because this paper is both detailed and long (75 pages) it is not included in this report.

Second I compiled the NAILSC study into English sentences so that normal people could understand it. This is included as Appendix F. It is actually a two section report, the first having just been described. The second section is an expansion of Navy Acronyms (e.g. NOR = Not Operationally Ready).

The third paper was a culmination of sorts since it is a proposed dialog of what that PLANES and BROWSER system should be capable of. This is Appendix B.

APPENDIX E: 3-M Summary Data and Typical Values for F-14A Aircraft

E.1. Monthly Summary Data by Tail Number

Indexed by ORG, PUC, TEC, BUSER

Date of Summary - (month - year)
Total flights in Month
Total Flight Hours in Month
Total Ship Based Flights
Total Ship Based Flight Hours
Readiness Reporting Hours

Not Operationally Ready Hours Due To:

1) Scheduled Maintenance
2) Unscheduled Maintenance
3) Supply (waiting for parts)

Reduced Mission Capability Hours Due To:

1) Scheduled Maintenance
2) Unscheduled Maintenance
3) Supply

Was The Aircraft With Its Assigned Unit During The Month

E.2. Daily Summary of Flights by Tail Number

Indexed by ORG, PUC, TEC and BUSER
Date of Summary - (Last Digit of Year + Day (1-365))
Ship or Land Based Flight
Flight Purpose
Departure Time

Flight Hours

Number Flights in Day

Number Catapult Takeoffs

Type of Landing (day/night and apparatus used in landing e.g. arrested landing)

Number of Landings

Was the Aircraft With Its Assigned Unit

E.3. Daily Summary of Maintenance by Tail Number

Indexed by ORG, PUC, TEC, BUSER

Date of Summary - (Last Digit of Year + Day (1-365)

Type of Maintenance - (NORMU, NORMS, NORS, RMCV, RMCS

Job Control Number of Maintenance - (may be multiple JCN done per day)

Work Unit Code (SYSTEM) of Aircraft Which is Being Repaired

Work Begin/End Time

Maintenance Hours

Supply Hours (waiting for parts)

Was the Aircraft With Its Assigned Unit

E.4. Maintenance Summary by Aircraft Series

Indexed by Work Unit Code (system or subsystem on which work is performed)

Nomenclature for WUC

Date - Time Period Report Covers (Arbitrarily)

Total Flight Hours

Total Maintenance Actions

Mean Flight Hours Between Failures

Mean Flight Hours Between Maintenance Actions

Failures Repairable in Place

Total Number of Failures

Total Unscheduled Maintenance Man Hours

Unscheduled Maintenance Manhour per Flight Hour

Maintenance Manhours per Maintenance Action

Maintenance Time per Maintenance Action

[Above Fields are Summed for Series Overall Total]

E.5. A Typical Summary for F-14A Aircraft

For a Six Month Period July '74 - December '74

Total Flight Hours - 9,886 hours

Total Maintenance Actions - 33,377

MFHBMA - .3 hours

In Place Repairs - 6,800

Total Failures - 12,530

MFHBF - .8 hr.

Unscheduled Manhours - 308,013 hrs.

Unscheduled Manhours/Flight hr. - 20.5 m/hrs/FH

Manhours per Maintenance Action - 6.1 m/hrs

EMT per Maintenance Action - 3.1 hr.
(average length of maintenance)

This is considered a reasonable aircraft

APPENDIX F*

Real-World Data on the 3-M Data Base

by

Forrest Conrad

Advanced Automation Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
January 1976

ABSTRACT

Based on a report of actual usage of the 3-M Data Base for Aircraft, Mechanicsburg, PA., this appendix provides information on:

(1) frequently investigated categories of concern,

(2) corresponding types of sentences the PLANES system must be able to answer, and

(3) a summary of the most frequently used abbreviations.

* This exactly reproduces Advanced Automation Group Working Paper 5.

F.1. CATEGORIES OF CONCERN (FREQUENTLY INVESTIGATED)

1. Determine the causes and actions required to correct high NOR rate equipment. 740/mo.

2. Determine whether to investigate an unsatisfactory material/condition report. 414/mo.

3. Respond to CNO request to justify grounding an aircraft. Determine whether incorporation/lack of incorporation of a technical directive was a factor in a crash. 6/mo.

4. Determine total maintenance accomplished on a given type/model/series during the past year, 1) at a given rework facility, 2) on a technical directive, 3) on aircraft rework, 4) on engine rework. 25/mo.

5. Determine technical directive status of specific bureau numbers (aircraft) for possible foreign military sale and/or determine TD status and past maintenance action history of equipment in response to foreign military query or sale. 4/mo.

6. Justify an engineering change proposal or evaluate benefits of a technical directive. 225/mo.

7. Determine whether to extend maintenance intervals. 50/mo.

8. Determine the committed workload on rework facilities. 12/mo.

9. Determine whether to extend an engine on MOT. 1/mo.

10. Determine when "wearout" on a type/model/series will require wholesale replacement. 74/mo.

Derived from NAILSC (Naval Aviation Integrated Logic Support Center), NALDA (Naval Air Logistics Data Analysis), Navair Airtask A4014013-0534-5401000001 APPENDIX C.

F.2. TYPES OF SENTENCES SYSTEM MUST HANDLE

The AN/ASN-90 inertial measurement set is number four on the aircraft degradation ranking summary for the period Oct. 73 thru March 74. Investigate and report on this problem.

The AN/ASN-90 inertial measurement set is fourth on the hit parade for Oct. 73 thru March 74. Investigate and report.

How many AWM, reason code, and hours were documented by VA-125 from Oct. 73 thru March 74 for WUC 73A50?

What BUNO's have had failures reported on WUCs 4614E00 and 1381000 over the last six months?

What is the OP RDY % for the last six months for VA-35 (F-14's) at Oceana?

What does it look like for all F-14 A/C?

What items (parts) are causing the rudder PC servo valve assembly to fail?

What is AIMD(ML-2) turnaround for AWP on part 137-JX?
for fault isolation
for repair
for validation 4/mo.

By part number and activity, give total items processed by AIMD last month with ATC = A
ATC = B/C/K/Z
ATC = 1
ATC = 4
ATC = 2, 3, 5, 8
ATC = 9 1/mo.

By part number and activity, give total items processed with ML = 2 last month where

no repairs were necessary	(ATC = A)
corrective action was required	(ATC = B/C/K/Z)
repair was not authorized	(ATC = 1)
there was a lack of parts	(ATC = 4)
AIMD limitations caused no action taken	(ATC = 2, 3, 5, 8)
the part was condemned	(ATC = 9)

1/mo.

How many manhours did VP-31 spend performing daily inspections during May 1973 - June 1974? 15/mo.

When corrective action was required (ATC = B/C/K/Z), what was the average days TAT by part number and activity?
TAT less AWP
average days AWP 1/mo.

What was the average AIMD TAT for ATC = A?
TAT for ATC = 1?
AWP for ATC = 4? 1/mo.

What was the AIMD ATC B/C/K/Z ML-2 EMT?
ATC 2, 3, 5, 8 TAT?

What was the average ML-1 MMH?
average ML-1 EMT? 1/mo.

Determine aircraft which had no flight hours during the last 30/60/90 days by BUNO and status code.

Find HANGAR QUEENS over the last 30/60/90 days by BUNO and status code. 1/mo.

Compare NORS and NORS percentages last month by squadron and by wing. 1/mo.

When was part 16357-XJ14 removed from BUNO 123456 because of bird strike damage?
What was the EMT, MMHR, AWM, AWP and AIMD associated with the repair?
Was the item BCM?
What was TAT, NARF and date of the RFI associated with the NARF repair? 4/mo.

How many TMS is part X 1479-3 installed on? 4/mo.

Determine cost per component of items BCM to depot. 8/mo.

Determine AWP for part X1479-3.
Determine reason for AWP for part x1479-3. 30/mo.

Has there been BCM actions for part x1479-3 in the AIMD?
Has there been no defect actions (ATC = A) for part X1479-3 in the AIMD? 180/mo.

What was the total number of cannibalizations by squadron and TMS during Jan. 1973 thru August 1973? 4/mo.

What were the manhours expended for corrosion prevention by squadron last month? 4/mo.

What was the number of ship board flights by TMS last month? 15/mo.

What is the monthly MMH/FH of F-4's vs. A-7's last year? 15/mo.

What is MFHBMA on A-7's by WUC last month? 4/mo.

Give verified failures and MMH by WUC on A-7's last month. 4/mo.

Give the number of items processed by VP-73 by squadron last year. 15/mo.

What is the avg. TAT for an item repaired at VP-136 during Jan. 1973 - June 1974? 4/mo.

What is the average number of items AWP in VP-137 last month? 4/mo.

What is the number of items repaired and replaced by TMS last month? 4/mo.

What were the reasons for failure per WUC per TMS last year? 4/mo.

How many hours were expended on calendar/phase inspections by ORG per TMS last month?
How does this compare to those listed on MRC DECK? 4/mo.

Define high maintenance serial numbers so they may be removed and overhauled. 4/mo.

Determine where inventory is by serial number. 8/mo.

Do a trend analysis on what type of failure of what parts have the highest repair rates. 30/mo.

What repairable components were installed on and removed from VAQ-128 aircraft BUNO 156859 during the months of Oct. and Nov. 74? 4/mo.

What are the serial numbers of those components having a WUC of 17740? 4/mo.

What parts or components caused a high assembly item to have a high failure rate in squadron VF-1 during Sept. 73? 4/mo.

Determine location of components/GSE/aircraft so as to better utilize the material. 31/mo.

Determine the number of removals per month by types and series of engines. 4/mo.

What is the reliability/utilization of part number XXXX? 4/mo.

Investigate component reliability failures by part number. 1/mo.

What was trend of VR-24 readiness in the past six months as opposed to personnel posture? 1/mo.

What were the monthly operationally ready percentages by TMS by squadron. 4/mo.

Determine if any changes have occurred in the failure rate of A/C components. 1/mo.

Identify the biggest problems with NOR hours.
man hours
degradation
maint. actions
aborts
UR's
mishaps 30/mo.

Determine the technical directive configuration of assigned aircraft by BUNO. 1/mo.

Did TDC 12345678901 solve the problem it was designed for? 4/mo.

Does A-7E BUNO 156875 have all applicable technical directives installed?
What A-7E aircraft do not have the most recent TD's incorporated? 4/mo.

Determine the effect of TD's in maintenance procedures on fleet workload. 11/mo.

What narrative info has been documented on the UR's against part QUUX-1732? 302/mo.

Which of the more than 900 unsatisfactory condition/material reports received per month are worth expending man hours to investigate? 900/mo.

What is the frequentcy of manfunction of part QUUX-1732 over all aircraft that incorporate it? 50/mo.

Is the quality of reworked products comparable to new material? 30/mo.

Compare the quality of reworked products by rework facility. 30/mo.

Compare maint. manhours of 'ace' to 'non-ace' aircraft.
failed parts
parts removed
mishaps
UR's 30/mo.

What fails and what are the high man-hour consumers at aircraft rework facilities by model/series and component? 1/mo.
Determine the average time between removals by type and series of engine. 1/mo.

How many maintenance actions for the last six months of data has PEI(VF-121) had with WUC 13153, 63150, 73A54?

What were the main reasons that WUC 73A50 malfunctioned in the A-7E aircraft (A/C) from Oct. 1973 through March 1974?

What WUC's at the five-digit level are causing maintenance problems on F-14 flight reference associated equipment (WUC-56X10)?

In the HF communications system on S-3 A/C, what WUC requires frequent maintenance?

Derived from NAILSC, NALDA, NAVAIR, Airtask A4014013-0534-5401000001 Appendices F and B.

F.3. COMMONLY USED ABBREVIATIONS

A/C	aircraft
ACT WC	action work center
ACT DATE	action date
ACTORG	action organization
AIMD	aircraft intermediate maintenance department
ATC	action taken code
AT	action taken
AWM	awaiting maintenance time
AWP	awaiting parts
BCM	beyond capability of maintenance
BUNO	bureau number
BUSER	bureau number/serial number
CANN.	cannibalization
CT	card type
ECP	engineering change proposal
EI	engineering investigation
EMT	elapsed maintenance time

FIIN	federal item identification number
FLT	flight
FP	failed part
FPC	flight purpose code
FSC	full systems capability
FSN	federal stock number
GSE	ground support equipment
HOWMAL	how the part malfunctioned
IP	installed part / items processed
ISS	issue / installation code
JCN	job control number
MA	maintenance action
MANHR	manhours
MAL CODE	malfunction description code
MC	meter code
MFG	manufacturer's code
MFHBF	mean flight hours between failures
MFHBMA	mean flight hours between maintenance actions
ML	maintenance level
ML1 REPAIRS	maintenance level one-repair effected locally

MMH	maintenance man-hours
MT	meter time
MTBF	mean time between failures
NFE	not fully equipped
NOC	not otherwise coded
NOR	not operationally ready
NORM	not operationally ready-maintenance
NORMS	not operationally ready-scheduled maintenance
NORMU	not operationally ready-unscheduled maintenance
NORS	not operationally ready-supply
NSC	naval safety center
NSN	national stock number
OR	operationally ready
ORG	organization
PN	part number
PUC/UIC	permanent unit/unit identification code
RFI	ready for issue
RMC	reduced material condition/reduced mission capability

RMCMS	reduced material condition scheduled maintenance
RMCMU	reduced material condition unscheduled maintenance
RP	removed part
RRC	reporting requirements code (for GSE)
RRS	readiness reporting status
SAF	support action form
SERNO (or SN)	serial number
SHPOP	ship operations (aircraft carrier landings, etc.)
SQDN	squadron
TAT	turn around time
TDC	technical directive compliance
TDSA	technical directive status accounting
TEC	type equipment code
TFH	total flight hours
TM	type maintenance code
TMS	type/model/series
TT	type of transaction
UR	unsatisfactory material/condition report
WC	work center code
WD	when discovered code
WDC	when discovered code
WUC	work unit code

REFERENCES

ACM 1971. CODASYL Data Base Task Group, April 1971 Report, (available from ACM).

Brown, J. S. and Burton, R. R. 1975. "Multiple Representations of Knowledge for Tutorial Reasoning," in Bobrow and Collins (eds.) Representation and Understanding, Academic Press, New York.

Bruce, B. 1975. "Case Systems for Natural Language," Artificial Intelligence 6 (1975), pp. 327-60.

Celce-Murcia, M. 1972. "Paradigms for Sentence Recognition," UCLA Dept. of Linguistics Memo.

Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks," Comm. ACM, Vol. 6, No. 3, June 1970.

Codd, E. F. 1971a. "Normalized Data Base Structure: A Brief Tutorial", Proc. ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Nov. 11-12, 1971.

Codd, E. F. 1971b. A Data Base Sublanguage Founded on the Relational Calculus", Proc. ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Nov. 11-12, 1971.

Codd, E. F. 1971c. "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, "Data Base Systems," New York City, May 24-5, 1971, Prentice-Hall, New York.

Codd, E. F. 1974. "Seven Steps to Rendezous with the Casual User," Proc. IFIP TC-2 Working Conf. on Data Base Management Systems, Cargese, Corsica, April 1974, North-Holland Publ. Co., Amsterdam.

Conrad, F. 1975. Unpublished internal communication.

Conrad, F. 1976. "BROWSER," M.S. Thesis (forthcoming), Dept. of Computer Science, University of Illinois, Urbana, October 1976.

Date, C. J. 1975. An Introduction to Database Systems, Addison-Wesley, Reading, Massachusetts.

Fillmore, C. 1968. "The Case for Case" in Bach and Harms (eds.), Universals in Linguistic Theory, Holt, Rinehart and Winston.

FMSOINST 4790.1A, Catalog of 3-M Aviation Information Reports, Dept. of the Navy, (Mechanicsburg, Pa., 1974).

Gabriel, R. P. and Finin, T. 1975. "The LISP Editor," Advanced Automation Group, Working Paper 1, Coordinated Science Lab., University of Illinois, Urbana.

Gabriel, R. P. and Waltz, D. L. 1974. "Natural Language Based Information Retrieval," Proceedings of the 12th Allerton Conf. on Circuit and System Theory, University of Illinois, October 1974, pp. 875-84.

Green, F. 1976. "Implementation of a Query Language Based on the Relational Calculus," M.S. Thesis (forthcoming), Dept. of Computer Science, University of Illinois, Urbana, October 1976.

IBM 1971. "Information Management System IMS/360, Application Description Manual," IBM Form No. H20-0524.

Malhotra, A. 1975. "Knowledge-Based English Language Systems for Management Support: An Analysis of Requirements," Advance Papers of the 4th Intl. Joint Conf. on Artificial Intelligence, Tbilisi, USSR, Sept. 1975, pp. 842-7.

Minsky, M. L. 1967. Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, N.J.

Moon, D. A. 1973. MACLISP Reference Manual, MIT Project MAC, Cambridge, Massachusetts.

NALDA/NAILSC (1974). "NALDA (Naval Air Logistics Data Analysis) System Data Requirements Determination Report," Naval Aviation Integrated Logistic Support Center, Patuxent River, Maryland 20670.

OPNAVINST 4790.1A, "Maintenance Data Collection Subsystem," The Naval Aviation Maintenance Program, Vol. III, Dept. of the Navy (Washington, D.C.).

Palermo, F. P. 1972. "A Data Base Search Problem," 4th Intl. Symposium on Computer and Info. Technology, Miami Beach, December 1972, Plenum Press; also available as IBM San Jose Research Report RJ1072.

Schank, R. C. 1973. "Identification of Conceptualizations Underlying Natural Language," in Schank and Colby (ed.) Computer Models of Thought and Language, W. H. Freeman.

Shannon, C. E. 1948. "A Mathematical Theory of Communication," Bell System Tech. Journal 27, 379-423, 623-656.

Smith, J. M. and Chang, P. Y. 1975. "Optimizing the Performance of a Relational Algebra Database Interface," Comm. ACM, Vol. 18, No. 10.

Tenczar, P. J. and Golden, W. M. 1972. "Spelling, Word, and Concept Recognition," Report, Computer-based Education Research Lab., University of Illinois, Urbana.

Waltz, D. L. 1975. "Natural Language Access to a Large Data Base: An Engineering Approach," in Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, U.S.S.R., Sept. 1975, pp. 868-72.

Waltz, D. L. 1976. "Natural Language Access to a Large Data Base," Naval Research Reviews, Vol. XXIX, No. 1, Jan. 1976, pp. 11-25. (Also reprinted in Computers and People, Vol. 25, No. 4, April 1976.)

Wilks, Y. 1975. "A Preferential, Pattern-Seeking, Semantics for Natural Language Inference," Artificial Intelligence 6 (1975), pp. 53-74.

Winograd, T. 1972. Understanding Natural Language, Academic Press, New York.

Woods, W. A. 1970. "Transition Network Grammars for Natural Language Analysis," Comm. ACM, Vol. 13, pp. 591-606.

Woods, W. A., Kaplan, R. M. and Nash-Webber, B. 1972. "The Lunar Sciences Natural Language System: Final Report," Bolt Beranek and Newman Inc., Cambridge, Massachusetts, Report No. 2378.