Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

*The research reported herein was partially supported by a grant from the National Science Foundation (GP-7069) and partially by USAF Proj. RAND (project #1116). Use of the University of Wisconsin Computing Center was made possible through support, in part, from the National Science Foundation and the Wisconsin Alumni Research Foundation (WARF) through the University of Wisconsin Research Committee.

A NET STRUCTURE FOR SEMANTIC INFORMATION STORAGE, DEDUCTION AND RETRIEVAL*

by

Stuart C. Shapiro

Technical Report #109

January 1971

ABSTRACT

This paper describes a data structure, MENS (MEmory Net Structure), that is useful for storing semantic information stemming from a natural language, and a system, MENTAL (MEmory Net That Answers and Learns) that interacts with a user (human or program), stores information into and retrieves information from MENS and interprets some information in MENS as rules telling it how to deduce new information from what is already stored. MENTAL can be used as a question-answering system with formatted input/output, as a vehicle for experimenting with various theories of semantic structures or as the memory management portion of a natural language question-answering system.

Keywords and phrases: question answering, memory net, memory structure, data structure, semantic memory, semantic information retrieval, deductive inference, fact retrieval.

1. INTRODUCTION

In order to develop machines capable of "understanding" natural language, it is extremely valuable, if not necessary, to design a method of organizing a corpus of data to facilitate the storage and retrieval of information on many subjects, some in depth, some in breadth; to facilitate the storage, retrieval and use of the many complex relationships among real-world concepts; to facilitate the storage, retrieval and use of information which tells how other information in the corpus may be used to further explicate implied relationships among concepts; and to facilitate the identification from the vast corpus of data of those pieces of information most directly relevant to any given topic.

This paper describes a data structure (MENS) and procedures for manipulating it (MENTAL) that have been designed to meet the requirements outlines above. This system is intended to be used as the memory of a natural language question answering machine (see [5; 6; 7; 8] and could also be used as the memory of a general theorem prover or problem solver. Since the system allows its user (either a human or an outside program) to specify the relations that will be used for the basic structuring of information, the system can be used for experimenting with data structures suitable for various contents and purposes. The major features of the data structure are:

It is a net whose nodes represent conceptual entities and whose edges represent relations that hold between the entities.

A distinction is made between n-ary relations about which information and deduction rules are to be stored and strictly binary relations that are used only to structure information about other entities. The former are represented by nodes in the net, just like any conceptual entity. The latter relations are the ones used as the edges of the net.

Some nodes of the net are variables, and are used in constructing general statements and deduction rules.

Each conceptual entity is represented by exactly one node in the net from which all information concerning that entity is retrievable.

Nodes can be identified and retrieved either by name or by a sufficient (though not necessarily complete) description of their connections with other nodes, likewise identified.

The system and data structure described here follow along the general lines laid out by such systems as Semantic Memory [9], TLC [10], Protosynthex II and III [12,17,18], GRAIS [3] and SAMEMAQ [15,16], but differ mainly in the clear separation of the two levels of relations and in the ability to store and use general deduction rules.

All the procedures for storing information into the data structure, as well as all those for explicit retrieval and some of those for implicit retrieval have been programmed in PL/1 and are running interactively on an IBM System/360. All the research reported herein has

proceeded both theoretically and by writing, checking out, revising and improving programs in PL/1, SNOBOL3 and Burroughs Extended ALGOL.

A more detailed discussion of MENS and MENTAL which also shows their applicability as an experimental vehicle is given in [14].

2. BASIC CONCEPTS OF THE STRUCTURE

The basic motivating factors for MENS were:

1. Unified representation: All conceptual entities about which information might be given and questions might be asked should be stored and manipulated in the same way.

2. Single file: All the information about a given conceptual entity should be reachable from a common place.

3. Multientried, converging search: A search of the file should start from as many places as possible and proceed in parallel, converging on the desired information.

4. Storage of deduction rules: Rules determining how deductions may be made validly, even when specific to certain areas or relations, should be stored in the memory file just like other information, and the system should be able to use them in directing its deductive searches.

5. Direct representation of n-ary relations: N-ary relations, for any n, should be as natural for the system as binary relations.

6. Experimental vehicle: The file should be designed without any commitment to a particular semantic theory, i.e. the memory system should be a research vehicle for experimentation on various ways of structuring the information in it.

In this section, we will describe how these motivating factors led to the particular structure decided upon.

Unified representation requires that every conceptual entity, i.e. every concept or individual about which one can talk, have a memory structure representation which can be put into relationships with representations of other conceptual entities. It further requires that all conceptual entities be represented in the same way regardless of their exact relationships to other conceptual entities. We will refer to a conceptual entity or to the logical representation of a conceptual entity as an item. When referring to the computer structure used to implement the representation of a conceptual entity, we will use the term item-block. In illustrations, we will picture an item block as a rectangle within which we will place an English word to indicate what concept the item block represents. If no such word exists some other symbol may appear so that the item block may be referred to. The full implication of unified representation is that every word sense, every fact and event, every relationship that is to be a topic of discussion between the system and its human discussant will be represented by an item. Therefore, the items must be tied together by relationships that are not conceptual entities. The reasoning for this is as follows. Statements (e.g., "Brutus killed Caesar.", "The sky is green.") are conceptual entities since we may say things about them such as someone believes them or they are false. Therefore, they must be represented by items, and such an item must bear some

relation to the items (Brutus, kill, green) that make up the statement. If this latter relation is a conceptual relation, the fact of this relationship's holding between two items may be discussed and thus must be represented by an item which then must have some relationship to that relation, etc. Eventually there must be some relation which is not conceptual, but merely structural, used by the system to tie a fact-like item to the terms partaking in it. We will refer to a conceptual relation as an item relation or simply a relation and to a non-conceptual relation as a system relation, link, or pointer. The MENS structure is, thus, a collection of items tied together by system relations into a directed graph with labelled edges. The nodes of the graph are the items and the edges are system relations. The edges are directed to indicate the order of the arguments of the system relation. The edges are labelled to allow for several different system relations. The distinction between item relations and system relations is very important and must be kept in mind.

Single file means that there will be exactly one item for each conceptual entity. Therefore, all the information about the conceptual entity will involve its item and be retrievalbe from its item block. Since the system relations are the links that tie items together and thus provide the information, this means that whenever a link goes from one item to another, there is an associated link in the reverse

direction. Looking at the fact and event items as records in a record oriented file and at the links going from participating items to fact and event items, MENS is an inverted file and may be searched as one. However, it is more than an inverted file, since links go the other way also. In illustrations, a link pair is represented by a line connecting two item blocks. The name of the system relation appears in the item block where the line emanates, from it. For example in Figure 1 the system relation AGENT goes from item 241/00010+23 to the item representing JOHN.

Multientried, converging search implies that items equally identifiable by the human conversant should be equally identifiable by the system. By this is meant that any item named by an English word can be located as quickly (by the same lookup procedure) as any other item so named, rather than some being locatable by lookup while others require an extensive search. Items that do not have English names, but must be identified by description will be located via searches that are quick or involved depending on the complexity of the description. The lookup is done through a dictionary which gives the internal names for the items which represent each of the senses of each natural language word used in the conversations. The internal name of an item is its address in secondary storage, so once looked up the item block is easily found. Items are connected to facts (which do not have

English names) as mentioned above and when two items are connected in the memory structure, each is reachable from the other since every link between two item blocks is stored in both directions. Another implication of multientried, converging search is that searching the file is done by starting at an arbitrary number of item blocks (all those that can be looked up directly) and converging to the desired information structures. This involves repeated intersections of sets of items as will be explained in section 3. Special care has been taken to make this search process as efficient as possible and special constructs have been developed for this purpose.

Storage of deduction rules implies that deduction rules* should be capable of being stored in and retrieved from the memory structure in the same way that specific information is stored and retrieved. This implies that the structures used to store deduction rules must be basically the same as those used to store specific information. It further implies that the executive routines must include a very general deduction rule interpreter that is capable of initiating searches of the memory and generating appropriate consequences based on any stored deduction rule.

---

*By "deduction rule" is meant any statement which, properly interpreted, provides information as to what statement(s) may be concluded from what other statement(s). Deduction rules include (among others) rules of inference of symbolic logic, general statements and disjunctive statements (any clause may be the conclusion if the negation of all the others holds).

Direct representation of n-ary relations implies that an item representing a relational statement based on an n-ary relation should have links to each of its n arguments directly, regardless of the value of n. This means that any item must be capable of having an arbitrary number of pointers emanating from it. This number may even change throughout the life of an item as the types of system relationships it has with other items change.

Experimental vehicle implies that the user must be given the capability of declaring what and how many system relations he will use rather than having a maximum number imposed on him. He must be able to decide what will be his conceptual entities rather than be provided with a closed set of them. He must be able to decide how items and pointers will be combined into structures to represent the information he wishes to work with. He must, finally, not be restricted as to what deduction rules the system may store and use.

3. EXPLICIT STORAGE AND RETRIEVAL

Both storage into and retrieval from MENS are accomplished by describing how an item is (or is to be) connected to other items in the net. The storage instruction in effect says, "Create an item and connect it into the net in this way." The retrieval instruction in effect says, "Tell me all items that are connected in the net in this way." Both instructions are expressed in a statement, called a specialist, which describes the item by describing the paths in the net that lead away from the item. These paths may be quite complicated, but the edges along the paths must be explicitly named system relations.

We will now describe the language which a human (or an external program) uses to interact with the system.

INPUT SYNTAX:

The input language is defined in modified BNF notation. Underlined words in lower case letters are non-terminal characters. Strings enclosed in square brackets are optional. Strings arranged vertically and surrounded by braces are alternatives -- one must be chosen. Strings followed by an asterisk may appear one or more times. Strings surrounded by broken brackets are informal English descriptions of object language strings. ⌀ represents a required blank; additional blanks may be inserted anywhere. The following characters are

delimiters in the language: , - ¬) (? . : ' = %.

A "character" is any legal character except a delimiter.

input → { relspec | (speclist) | . | describe-request }

describe-request → DESCRIBE ∅ cname [{ ∅ , } cname]*

relspec → $ linkname ∅ { S M } linkname ∅ { S M }

speclist → spec [-spec] [¬ restrictions] [= vname]

restrictions → (linkname [, linkname]*)

linkname → <a string of 1 to 13 characters>

spec → { buildspec | findspec }

buildspec → (. linkname : speclist [, linkname : speclist]*)

findspec → { name | (name [, name]*) | (findprefix linkname : speclist [, linkname : speclist]*) }

findprefix → ? num , num { (vname) , }

name → { cname ; 'vname ; %vname }

cname → { < a string of 1 to 13 characters > ; < 3 digits > / < 5 digits > + < 3 digits > }

vname → { < a string of 1 to 13 characters, the first of which is not X, Y, or Z > ; < a string of 1 to 13 characters, the first of which is X, Y, or Z > }

num → { < any integer i, $0 \leq i \leq 2^{31}$ > ; # }

Input Semantics

A relspec is used to declare a system relation, i.e. a relation that will be used as a pointer in the file. The first linkname in a relspec will be the symbolic name of a pointer (considered the forward pointer of the system relation) and the second linkname will be the converse of the first (and will be called the reverse pointer). Each pointer will be single or multiple depending on whether "S" or "M" follows its linkname. Examples of relspecs are:

$ AGENT S *AGENT M

$ VERB S *VERB M

A cname is the external name of an item in the net. The first form of a cname (a string of 1 to 13 characters) is the one normally used in a speclist, and is introduced by the user to represent some concept he wishes to discuss. Although we will use English words for these cnames in this paper, it must be remembered that they each stand for an unambiguous concept (word sense). A cname is associated with an item after the first time it is used in a speclist, and maintains that association. The second form of a cname is the direct representation of the internal name of an item and is used by the system to mention to the user an item that does not have another external name. The user should not use such a cname unless it has previously been used by the system in the reply to a speclist or in the display following a describe-request. Examples of cnames are:

JOHN	241/00010+23
HAS_SENSE_1	240/00023+002

The describe-request causes the system to display, for each cname in the request, all paths of length 1 emanating from the item identified by the cname. That is, for each item identified, all pointers emanating from it are listed, and with each pointer is listed all items pointed to. An example of a describe-request is:

DESCRIBE JOHN,241/00010+023,LOVES 241/00023+002

A possible system response to this request is:

DESCRIPTION OF JOHN:

*AGENT	241/00010+023
*OBJ	241/00010+024
	241/00023+002

DESCRIPTION OF 241/00010+023

AGENT	JOHN
VERB	LOVES
OBJ	JANE

DESCRIPTION OF LOVES

*VERB	241/00010+023
	241/00010+024
	241/00023+002

DESCRIPTION OF 241/00023+002

AGENT	SUE
VERB	LOVES
OBJ	JOHN

A structure described by this response is shown in Figure 1 .

A vname is a variable of the input language. It may be associated with a single item or with a list of items, but when "." is given as the input (not enclosed in any parentheses) all vnames lose their

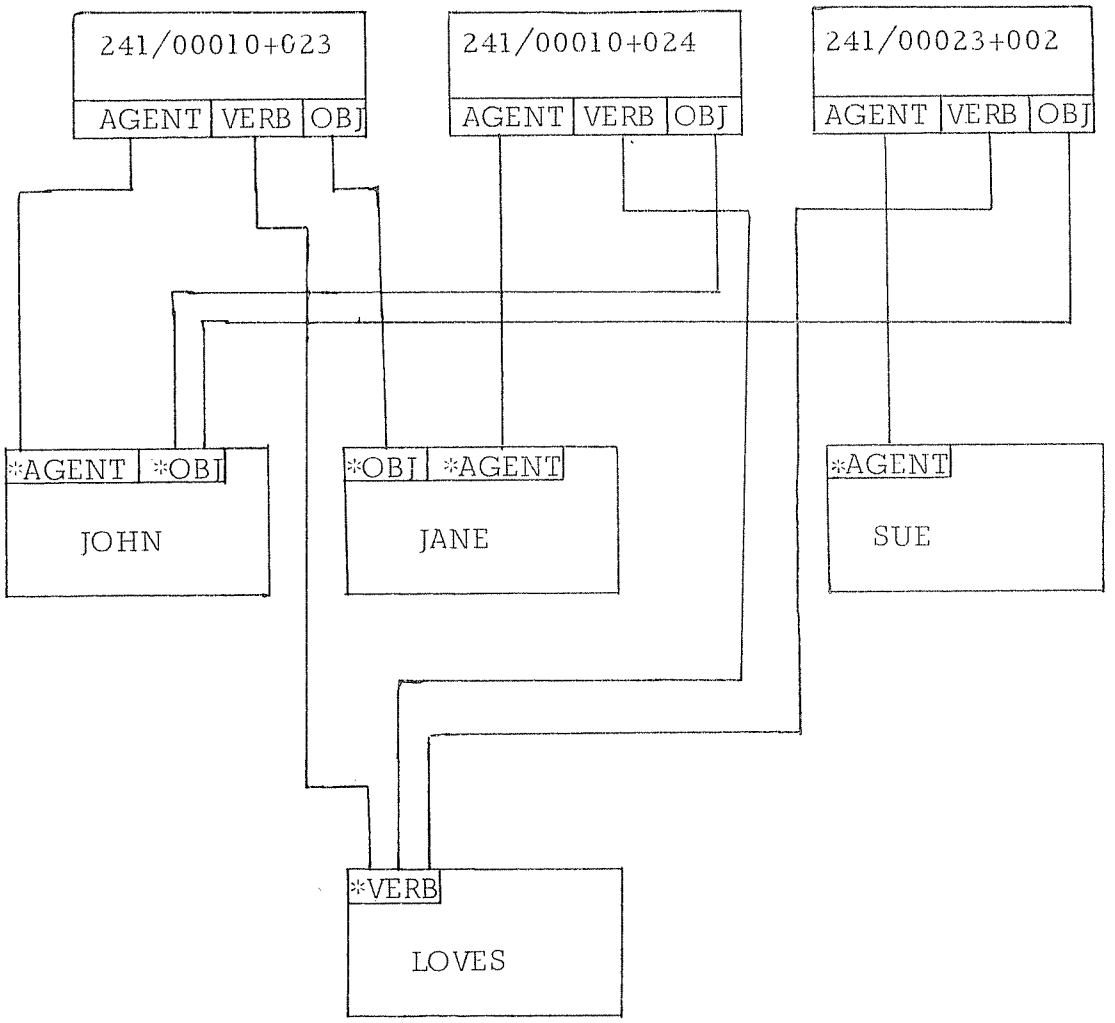

Figure 1. A MENS substructure, described in the text.

associations. It is important to distinguish between variable items and vnames. Variable items (see section 4) explicitly exist in the net, although they do not have external names. There are also some constant items that do not have external names, for example items which represent facts or events. Vnames may stand for either of these two types of items, and may also stand for items which do have external names. The important thing about a vname is that its association with an item or a list of items is only temporary. Furthermore, the system never uses a vname to refer to any item; it is used only by the user. The only time the distinction between vnames that begin with X, Y or Z and those that do not matters is when the first appearance of a vname immediately follows the delimiter " ' ". In that case a new item will be created in the net and the vname will temporarily be assigned as its name. If the vname begins with X, Y or Z the item created will be a variable item. Otherwise, the item created will be a constant item that will not have an external name. Although the vname construct is not the only way to introduce a constant item without an external name, it is the only way to introduce a variable item into the net. If the first use of a vname is in a findprefix or immediately preceded by "%", or in the = option in a speclist a new item will not be created, but the vname will be assigned an item or a list of items which will be found in the net according to the instructions embodied in the speclist.

The speclist is used both for storing new information into the net and retrieving information from it. Its main component is the spec, which is considered to have as its value a list of zero or more items. If a speclist consists of only the spec, the value of the speclist is the value of the spec. If the -spec option is included, the items on the list that is the value of that spec are removed from the value of the speclist. This allows a retrieval request of the form: "Tell me all items described in $spec_1$ that are not also described by $spec_2$." For example, the request to list all things written by Scott except Ivanhoe might be:

((?0,#,*OBJ:(?0,#,AGENT:SCOTT,VERB:WHITE))-IVANHOE)

The ¬ restrictions option causes the removal from the value of the speclist of any item that has any of the links named by the linknames of the restrictions emanating from it. The purpose of this option is to limit the value of the speclist to items without certain extraneous pointers. For example if a TIME link were used to point from items respresenting events to items representing the time interval of their occurrence, and the TIME link were not to emanate from any item representing a "timeless fact", then a request for all items representing timeless facts about the United States might appear as:

((?0,#,AGENT: UNITED_STATES) ¬ (TIME))

The -spec and -restrictions options would not, of course, be used in

a speclist whose initial spec is a buildspec. The =vname option causes the vname to be assigned as its temporary value the item or list of items that is the value of the speclist. The main use of this option is that if the same speclist is to appear in one spec in more than one place, much retrieval time can be saved if the =vname option is used in the first occurrence of the speclist and the other occurrences of the speclist can be replaced by the vname in the 'vname form of a name. For example, a retrieval request for all those who both love and are loved might be:

```
(?0,=,*AGENT:(?0,#,VERB:LOVE)=LUV_RELATIONS,*OBJ:
    'LUV_RELATIONS)
```

The spec is the basic construct for describing items to be built or found in the net. The item(s) described by the spec is the value of the spec and the procedures used to evaluate the spec are the main storage and retrieval procedures of the MENTAL system. There are two ways an item can be described: by its name or by a description of its connections within the net structure. Use of the item's name references the item directly -- the internal name is either a direct translation of the name or is discovered by lookup in the main symbol table or in the temporary vname symbol table. Use of the description form requires searching the net. The description is formed in the following way. Suppose you are looking at the actual item block. List the pointers

that emanate from the item block, and for each pointer list all the item blocks it goes to. These item blocks are listed by either giving their names or describing them in the same way as the original item is being described. At least one pointer of such a second level item points back to the original item, viz. the converse pointer of the pointer that points from the original item, so it will clarify nothing to list it. It may be the case, however, that some other item is encountered more than once in this expanding description. In that case, if its name is not known, the %vname option is used as mentioned above and described below to insure that this significant property of the structure is represented in the description. The description is continued until all paths that lead away from the original item being described end in an item described by a name. What has thus been described is a sub-structure of the net structure, and at the center of the sub-structure (or, we may say, at the head of the sub-structure) is the item described by the spec. It may be that the description fits more than one sub-structure of the net. In this case, the value of the spec is a list of all items that are heads of the sub-structures so described. If no sub-structure fits the description, the value of the spec is the null list. If the spec was a buildspec, a new item would be connected into the net so that it would be the head of a substructure described by the spec, and the new item would be the value of the spec. In describing an item it is not necessary to list all pointers emanating from

it if some are not known or if their existance is irrelevant for the intended retrieval.

We will now consider the findspec in more detail, specifically those in which the findprefix occurs. The first num is the minimum number of items which are to be found satisfying the description, while the second num is the maximum number. If the number of items found is less than the first num, more must be found using the deduction techniques (see sections 4-9). If the number of items found exceeds the second num, there is a semantic ambiguity that must be cleared up. The character "#" is used to represent the largest integer that can be held in the internal computer field used to store the nums. Some uses of the nums are:

If the spec is a definite description -- ?1,1

To find the active members of a football squad -- ?40,40

To find all the authors of a coauthored book -- ?2,#

The (vname) option is a way to change the value of a findspec from the head of the substructure described by the findspec to an item which occurs elsewhere in the substructure. This is sometimes necessary when the complexity of the substructure precludes the item being sought from bing described in the normal way. For instance if we wanted a list of all narcissistic people, we might be tempted to use one of the following equivalent findspecs:

(?0,#,*AGENT:(?0,#,VERB:LOVE),*OBJ:(?0,#,VERB:LOVE))

(?0,#,*AGENT:(?0,#,VERB:LOVE)=S,*OBJ:S)

Each of these specs would, however, evaluate to a list of all those who love and are loved, not necessarily by themselves. The proper way to form the request would be:

(?0,#(N)AGENT:'N,VERB:LOVE,OBJ:'N)

Similarly, the proper request for all who love someone who loves them back would be:

(?0,#(LOVED_ONE)AGENT:'LOVED_ONE,VERB:LOVE,OBJ:
(?0,#,*AGENT:(?0,#,VERB:LOVE,OBJ:'LOVED_ONE)))

The manner in which such a spec is evaluated is discussed below. We will now discuss how a spec that does not contain a vname is evaluated.

First let us look at the simplest spec -- where all speclists in the spec are just names. Say we wish to enter into the net the sentence, "John kissed Mary in Chicago on Tuesday." and we want it to have the structure described in the buildspec:

(.AGENT:JOHN,VERB:KISS,OBJ:MARY,LOC:CHICAGO,
TIME:TUESDAY)

We would enter an input consisting of the above speclist in an additional pair of parentheses. The linknames and external names would be looked up in the appropriate symbol tables and a new item block would be designated to represent the sentence. Note that every buildspec causes a

new item block to be built. The rationale for this is that every buildspec is supposed to represent a conceptualized piece of information about which further information might be given. For example, the above example might, in fact, be part of the sentence, "Henry said, 'John kissed Mary in Chicago on Tuesday.' " Furthermore, no check is made to determine if there is already an item in the net which is described by the buildspec. Although it was at one time planned to use an already existing item whenever it satisfied a buildspec instead of building a new one, it was eventually realized that this involved certain problems. One problem is whether two instances of a sentence reporting an event are two reportings of the same event or reportings of two similar events. Also, if the same item were used for the sentence represented by X in the sentences, "Henry said X." and "Bill said X.", the sentence "John heard what Henry said." would imply "John heard what Bill said." which would not necessarily be correct. Therefore, it has been left to the user (be it a human or a parsing-transducing program) to ascertain if a given substructure already exists and if so, whether or not to reuse it.

The item created for the above buildspec is given an AGENT pointer to the block representing JOHN, a VERB pointer to the block representing KISS and so on, so that it has five pointers emanating from it. The block representing JOHN gets a pointer for the converse

of AGENT (say *AGENT) pointing from it to the block representing the statement. Presumably, *AGENT has been declared in a relspec to be a multiple pointer. In that case JOHN may already have *AGENT pointers to other item blocks, and the name of the new block will be added to a *AGENT multiple pointer list. If *AGENT was declared to be a single pointer and JOHN already had a *AGENT pointer to another item block, the attempt to add another *AGENT pointer to JOHN will be in error and will not be performed.

To enquire if the sentence, "John kissed Mary in Chicago on Tuesday." is already in the net, the following input would be entered:

```
((?0,#,AGENT:JOHN,VERB:KISS,OBJ:MARY,LOC:CHICAGO,
   TIME:TUESDAY))
```

This is a request to list the names of any (zero or more) items which have the named system relationships to the named items. The item created for the above example would be an answer to this request and so would any other item that had these pointers, even if they also had additional pointers. The items would be retrieved in the following way. The list of items pointed to by the *AGENT pointer (let us assume that we have declared all linknames so that the linkname for the converse pointer is the linkname for the forward pointer with "*" prefixed) from the JOHN block is retrieved, along with the list of items pointed to by the *VERB pointer from the KISS block, the list of items pointed

pointed to by the *OBJ pointer from the MARY block etc. These lists would be intersected and the result would be the value of the findspec and the answer to the input request. The methods making possible efficient intersecting of these lists are derived from the algorithms for list set generators [13].

The situation is slightly more complicated when the embedded speclists are descriptions. First the embedded speclists are evaluated leaving a findspec of the form (...L:(I_1 I_2 ...)...). Since we are looking for an item with an L pointer to at least one of the I_1 or I_2 or ... and likewise for the other linknames in the findspec, what we want to intersect are the union of the *L lists from each of the I's with the unions of the other converse pointer lists. This intersection of unions is performed efficiently using the methods discussed in [13]. If this spec were a buildspec, an item would be created with an L pointer to each of the I's. If L were a single pointer but more than one I appeared in the list, an improper substructure would be built, so it is important when building an item with a single pointer to an item which is to be found, that the findspec describing the item to be found have the definite descriptor (?1,1) notation.

We will now discuss the evaluation of a spec that contains vnames. As was mentioned above, if the first occurrence of a vname is preceded by the delimiter ', it is immediately assigned a new item. From then until

the appearance of the input ".", every occurrence of that vname is immediately replaced by the name of the item which has been assigned to the vname. Therefore, we are now concerned only with vnames whose initial occurrence is preceded by the delimiter "%" or whose initial occurrence is in the findprefix. The prupose of such a vname is to specify that some unknown block is reachable by several different paths from the head of the substructure described by the spec. Thus, such vnames should be used in findspecs rather than in buildspecs. When a spec with vnames is evaluated, associated with every item in the value is a substitution which is a list of every vname in the spec and with each vname the item(s) that are the value of the vname as determined by the evaluation of the spec. Thus, if two embedded speclists in a spec both use the same vname, when their values are intersected the substitutions within them are compared and adjusted so that every path specified by the position of a given vname in the spec leads to the same item.

The intersection and union routine have been modified to take substitutions into account as well as to update them properly as the intersection and union operations are proceeding. Further discussion of this in this paper is precluded by space limitations. A detailed analysis is given in [14].

4. REPRESENTATION OF DEDUCTION RULES

In section 3 it was shown how the MENS structure is used for explicit storage and retrieval. In this section we will explain how it can be used for deduction. Since storage of deduction rules is a motivating factor of this project, the deduction method will involve the storing of general deduction rules and the use of fairly simple theorem proving techniques. The reason for this is that we want the system to be as general as possible and we want to concentrate on the data structure rather than the executive routines. It would be possible to build a complex and sophisticated theorem prover which uses MENS for its data storage, but this is not our current interest.

In order to allow for complete generality in what deduction rules could be stored, including arbitrary orderings of arbitrarily many quantifiers, it was decided to represent quantifiers and variables directly in the structure, and build executive routines to interpret the deduction rules. These routines would operate, upon being given a deduction rule, by carrying out searches required by the rule and building consequences justified by the rule. Representing quantifiers and variables directly seems to be a compromise of the motivating factor of unified representation since they will require special routines to deal with them and their status as conceptual entities is questionable.

However, dealing with the order of quantification implied by some English sentences is enough of a problem that at least one linguist believes that quantifiers and variables might profitably be comprehended by the base rules of English grammar [1, p. 112]. Besides, including this capability extends the use of the system as an experimental vehicle, another motivating factor.

The decision to allow direct representation of variables leads to the questions of how to represent them and what will be allowed to substitute for them. Considering the second question, the conclusion is that a variable should be able to stand for any item but not for any system relation. This is supported by the discussion in section 2 that anything about which information could be given should be represented by an item, that all items should be equally able to have information stored about them, and that system relations could not have information given about them since they are not conceptual entities. As Quine says, "The ontology to which one's use of language commits him comprises simply the objects that he treats as falling ... within the range of values of his variables." [11, p. 118 quoted in 2, p. 214]. Since the ontology of the data structure comprises the set of items (by definition of item), the values of the variables must be allowed to range over all the items, and since the system relations are to be ex-

cluded from the ontology, not allowing them to substitute for a variable reinforces their exclusion. Allowing the variables to range over all the items, however, brings up the possibility of storing the paradoxes that were eliminated from formal languages only with the introduction of types of variables or restrictions on assertions of existence (of sets). This possibility will be accepted. We make no type distinctions among the items and impose no restraints on item existence, leaving the avoidance of paradoxes the responsibility of the human informant. We will do the same with the variables. However, we do use restricted quantification. What is meant by this is that with each quantifier in a deduction rule will be included, not only the variable it binds, but also an indication of the set of items over which the variable ranges. Woods [19] uses restricted quantification to reduce the time needed to handle a request by including in the restriction a class name and a predicate. The class name must be of a class for which there exists a generator that enumerates all the members of the class one at a time. Each member is tested with respect to the predicate. Those for which the predicate is true are acted on by the main body of the request. Our restrictions may be more general. We will allow any statement, however complex, about the variable. This statement will be used as a search specification to find all items in the structure for which the statement is true. The

set of such items will comprise the range of the variable. Thus, even omega ordered type theory may be represented in the structure by entering a statement about every item giving its type and including type specifications in the restrictions on each variable.

We now return to the question of how variables should be represented. Each variable will be represented by its own item block. All occurrences of the same variable within a given deduction rule will be represented by the same item and no such item will be used in more than one deduction rule. The same item is used for all occurrences of a variable in a deduction rule so that a substitution made for the variable in one occurrence will at the same time be made in the others and so that all the information about what items can substitute for the variable will be reachable from one place. Different items are used in different deduction rules to eliminate the possibility of information about a variable in one deduction rule becoming associated with a variable in another. Part of the internal name of an item is used to distinguish variable items from constant items so that an item can be recognized as a variable when it is pointed to from another item.

Besides quantifiers and variables, the connectives NOT, AND, OR, IMPLIES, IFF and MUTIMP* are also represented as item relations

---

*MUTIMP stands for mutual implication. It is a predicate with an arbitrary number of arguments and says that its arguments mutually

in the structure and the executive routines that interpret the deduction rules are designed to handle them.

Deduction rules are stored using two types of items that will be recognized by the executive routines. We will call them quantifier clauses and connective clauses. A quantifier clause is the head of a quantified general statement and has four special systems relations emanating from it. They are:

(i) C points to the quantifier

(ii) VB points to the variable being bound

(iii) R points to the restriction on the variable

(iv) S points to the scope of the quantifier

A connective clause is the head of a construction formed of several clauses joined by one of the connectives mentioned above. It has an OP system relation to the connective and one of the following sets of argument relations:

---

*(continued) imply each other by pairs (are pairwise equivalent). Looked on as a binary connective, MUTIMP, like AND and OR and unlike IMPLIES and IFF is idempotent as well as associative and commutative. A possible definition of MUTIMP is:

$$\text{MUTIMP}(P_1,\ldots,P_n) \stackrel{\text{df}}{=} \underset{i=1}{\overset{n}{\text{AND}}} \left(P_i \text{ IMPLIES } \underset{\substack{j=1 \\ j\neq i}}{\overset{n}{\text{AND}}} (P_j)\right)$$

That is if MUTIMP $(P_1,\ldots,P_n)$ is true and P_i is true (false) for some i, $1 \leq i \leq n$, then P_i is true (false) for all i, $1 \leq i \leq n$. For two arguments MUTIMP is equivalent to IFF.

(i) ARG to the argument if the connective is unary (NOT)

(ii) ARG1 to the first argument and ARG2 to the second argument if the connective is binary (IMPLIES,IFF)

(iii) MARG to all the arguments if the connective is associative, commutative and idempotent (AND,OR,MUTIMP)

The clauses forming the arguments of a connective clause and those forming the restriction and scope of a quantifier clause may be any net sub-structure with the requirement that a clause may contain a free variable only if a sequence of converse argument pointers, converse restriction pointers and converse scope pointers leads to a quantifier clause in which that variable is bound.

Examples of deduction rules are given below. Each deduction rule is given first as an English language statement and then as a buildspec.

1. Every man is human.

(.Q:ALL,VB:'X,R:(.AGENT:'X,VERB:MEMBER,OBJ:MAN),
S:(.AGENT:'X,VERB:MEMBER,OBJ:HUMAN))

2. Every car has-as-part an engine.

(.Q:ALL,VB:'X,R:(.AGENT:'X,VERB:MEMBER,OBJ:CAR),
S:(.Q:EXISTS,VB:'Y,R:(.AGENT:'Y,VERB:MEMBER,OBJ:ENGINE),
S:(.AGENT:'X,VERB:HAS_AS_PART,OBJ:'Y)))

3. If a male is the child of someone, he is the sone of that person.

```
(.Q:ALL,VB:'X,R:(.AGENT:'X,VERB:MEMBER,OBJ:MALE),
S:(.Q:ALL,VB:'Y,R:(.AGENT:'X,VERB:CHILD_OF,OBJ:'Y)
S:(.AGENT:'X,VERB:SON_OF,OBJ:'Y)))
```

4. John is at home, at SRI or at the airport.*

```
(.OP:OR,MARG:(.AGENT:JOHN,VERB:AT,OBJ:JOHNS_HOME),
        MARG:(.AGENT:JOHN,VERB:AT,OBJ:SRI),
        MARG:(.AGENT:JOHN,VERB:AT,OBJ:AIRPORT_4))
```

---

*This sentence taken from Green and Raphael [4].

5. USE OF DEDUCTION RULES

There are six operations that can be performed with respect to a deduction rule in MENS. They are:

(i) It may be used for generating consequences.

(ii) It may be confirmed by exhaustive induction, i.e. proved F-true in the universe of the data structure at any given time.

(iii) It may be deduced from other deduction rules.

(iv) It may be refuted by finding a counter-instance in the data structure.

(v) Its negation may be deduced.

(vi) It may be treated as a specific statement, which includes its use as an assumption in the deduction or negation of other rules as in (iii) or (v).

We will be mainly interested in using a deduction rule for generating consequences. In this process, there are two ways of using a restriction. They are:

(i) A possible substitution for the variable may be checked to see if it fulfills the restriction.

(ii) The data structure may be searched to find all items that fulfill the restriction.

The amount of information that may be deduced with any deduction rule depends on more than the quantifiers and the number of items

that are found able to fulfill the restrictions. It also depends on the structure of the logical connectives in the deduction rule. For example, a deduction rule might have a consequent that was the conjunction of several sub-structures. Thus, several independent sub-structures might be deduced from each choice of items to substitute for the variables. There are, therefore, several different ways we may use a deduction rule for generating consequences. We may instantiate over all items that satisfy the restrictions or just over those we are interested in. Similarly, we may generate all the consequences justified by the deduction rule or just those needed to answer a particular question.

In the following sections, we will first discuss how a deduction rule useful for answering a particular question is found, and then discuss how the executive routines interpret the deduction rules and generate consequences.

6. FINDING DEDUCTION RULES

A deduction rule is needed when the number of items found to satisfy a findspec (see section 3) is less then the minimum number required. The problem then, is to find a deduction rule capable of generating an item that satisfies the findspec. Say the findspec is

(i) $(?0,\#,L_1:(I_{11},\ldots,I_{1m_1}),\ldots,L_n:(I_{n1},\ldots,I_{nm_n}))$

where $L_1,\ldots,L_n$ are system relations and $I_{11},\ldots,I_{nm_n}$ are specific items (we will assume at first that a substructure only one level deep is required and consider the case of several level structures later). In order for a deduction rule to generate the desired item, it must be headed by a quantification clause that is connected through a path of scope and argument pointers to an item which contains the labels $L_1,\ldots,L_n$ one or more of which point to variable items and the rest of which point to some item in the appropriate list in (i). We can, therefore, locate the deduction rule by searching for any item that satisfies the find-spec:

(ii) $(?0,\#,L_1:((I_{11},\ldots,I_{1m_1}) \cup V),\ldots,L_n:((I_{n1},\ldots,I_{nm_n}) \cup V))$

where V is a list of all variable items in the data structure. For the sake of efficiency, we maintain an item which we shall call VBL. No other item in the structure contains a pointer to VBL, but whenever a variable item contains a pointer L to an item I, VBL also contains a pointer L to the item I. Thus, the find-spec (ii) is equivalent to:

(iii) $(?0,\#,L_1:(VBL,I_{11},\ldots,I_{1m_1}),\ldots,L_n:(VBL,I_{n1},\ldots,I_{nm_n}))$.

Note that any item that satisfies (i) must be an instantiation of any item that satisfies (iii) and, further, it is possible to deduce an item that satisfies (i) only if an item satisfying (iii) exists in the data structure.

For each item, I, found satisfying (iii) we may record what substitutions we are interested in for the variables pointed to from I. If I has a pointer L_i to a variable item X_i, we record that the only items we are interested in substituting for X_i are $I_{i1},\ldots,I_{im_i}$ by putting them in a "possible substitution" list for X_i. They will later be checked against the restriction on X_i.

For each item I satisfying (iii), we then follow the paths of reverse scope and argument pointers until coming to an item D that is the head of a deduction rule. This will be a deduction rule capable of generating the consequence we are interested in. While following this path a <u>trace list</u> is created. This is a list $(S_1,\ldots,S_k)$ where S_k is I, S_1 is pointed to by a scope or argument pointer from D, and S_i, $2 \leq i \leq k$, is pointed to by a scope or argument pointer from S_{i-1}. The trace list will be used to limit the consequences generated to the ones desired.

In the case of failing to find items matching a <u>findspec</u> involving several levels, the same process is carried out, but we must be sure

to allow for all possibilities of variables replacing constants. That is, each level is handled as above for progressively higher levels, and the reverse scope and argument pointers are not followed until the highest level has been done.

7. GENERATE

The routine to generate consequences from a deduction rule is a recursive procedure that is initially given the internal name of an item that heads a substructure with no free variables. It returns a list of items (internal names) that head substructures representing the consequences that have been generated. These substructures might then either be left in the data structure or be erased. The Generate routine is written to generate consequences according to the author's understanding of the meanings of the quantifiers and logical connectives. It is not designed to prove theorems, but to use deduction rules and other data that have been stored and are assumed to be valid by generating consequences of them.

The various sections of the Generate routine assume the existence of certain global information, viz:

(i) If the trace flag (TRFL) is set, a trace list (TRACELIST) has been built as described above.

(ii) The negation flag (NEGFL) is used to pass negations down to minimum scope. If it is set the substructure being considered should be considered to be the negation of itself.

(iii) Every variable item has a list of possible substitutions, a list of substitutions and a substitution. The list of possible substitutions may be filled as described above. The list of substitutions

consists of those possible substitutions that have survived a check against the restriction or those items that have been discovered to fulfill the restriction via a search on the data structure. The substitution is the item actually substituting for the variable at a given time during generation.

As examples, two sections of the GENERATE routine will be described - ALLGEN, which generates a substructure headed by the universal quantifier, and ORGEN, which generates a substructure whose main connective is OR.

ALLGEN transfers to EXISTSGEN (just after the point where EXISTSGEN tests NEGFL) if NEGFL is on. Otherwise, if there is a list of possible substitutions for the variable of the quantifier clause, they are checked against the restriction and those that succeed are placed in the substitution list. If there is no list of possible substitutions, the restriction is used to direct a search for all valid substitutions and they are put in the substitution list. For each item in the substitution list as the substitution for the variable, GENERATE is called recursively with the scope of the quantifier clause as argument. If TRFL is on, TRACELIST is popped before GENERATE is called since the top item on it must be the item pointed at via S.

ORGEN transfers to ANDGEN (after its NEGFL test) if NEGFL is on. Since we want to generate the most concise information possible, an attempt is made to refute each item pointed to via MARG (except for the one on top of TRACELIST if TRFL is on). If only one item is not refuted, it is generated. Otherwise a disjunction is generated with the instantiation of each item that was not refuted as a disjunct. If ORGEN was transferred into from ANDGEN, NEGFL will be on and items will be discarded if confirmed rather than refuted. If ORGEN was transferred into because NEGFL was set at MUTGEN, two passes through ORGEN will be made, one with NEGFL on and one with it off.

8. CONFIRM AND REFUTE

The Confirm and Refute routines are used by the Generate routine as indicated above. It is, of course, possible for an expression to be neither confirmed nor refuted. For example, the statement "All men have two arms" would be neither confirmable nor refutable if we knew of exactly 100 men, of whom 99 had two arms, but we had no information about the hundredth.

The Confirm and Refute routines also use the author's knowledge of the quantifiers and connectives. For example a disjunction is confirmed if and only if any argument is confirmed and is refuted if and only if all the arguments are refuted.

9. SUBSTRUCTURE DIRECTED SEARCHING

Using a restriction to find all the items that satisfy it and finding an instantiation of a substructure containing free variables in order to confirm or refute it require a process similar to the one used to find an item described by some findspec. Such general substructures may contain some items which are connected to the head item by several different paths. If these items are constant items, any instantiation of the general substructure will contain them at the end of similar paths from the head item. If, however, they are variable items or items heading substructures containing variable items, the instantiation substructures will have different items in their place and we must be sure that no item in the general substructure is substituted for by more than one item in any instantiation substructure. This is done in the same way as evaluating a findspec which contains vnames which originally appeared preceded by "%" or in the findprefix.

10. SUMMARY

In retrospect, we can see several significant facets of the MENS structure and the MENTAL system. First, the work has been developed with a unified viewpoint grounded in the theoretical basis represented by the six motivating factors discussed in section 2. Underlying these have been the desires to maintain complete generality and to keep the executive routines as simple and general as possible. Thus the number of ad hoc features have been kept to a minimum. The only departure from building just a structure and those routines necessary to manipulate the structure ignoring what information might be stored in the structure was the establishment of the system relations and item relations used to store deduction rules and the executive routines to interpret them. Once that was done, however, no further constraints were placed on the deduction rules so that generality was maintained to a large degree.

Another significant facet of MENS is the two levels of relations -- system relations and item relations. System relations are the basic organizational mechanism of the structure, yet the user is allowed to define the ones he wants to use and thus may experiment with different semantic structures. Item relations are the conceptual, meaningful relations that hold between other concepts, yet the fact that they are relations is preserved only by the way they are connected in the struc-

ture, which is determined and interpreted by the user. Item relations, as conceptual entities, may have stored information about them as well as information using them.

A very important facet of MENS and MENTAL is the ability to enter, retrieve and manipulate deduction rules the same way specific facts are entered, retrieved and manipulated, yet deduction rules are used by the system to deduce information that was not previously explicitly stored in the structure. Thus one may explain to the system what a concept means by giving, in general terms, the implications of the concept, and one may give this explanation just like he gives the system any other information.

The system and structure as presented in this paper provide an environment in which important problems in question-answering and conputer understanding may productively be investigated. Also MENS and MENTAL may be used as an experimental vehicle for further research in semantic structures.

REFERENCES

1. Bach, E. Nouns and noun phrases. Universals in Linguistic Theory, Bach, E. and Harms, R. T. (Eds.), Holt, Rinehart and Winston, New York, 1968, 90-122.

2. Carnap, R. Empiricism, semantics, and ontology. in [10], 205-221. Originally in Revue Intern. de Phil. 4 (1950) 20-40.

3. Elliott, R. W. A model for a fact retrieval system. unpublished Ph.D. dissertation, University of Texas, Austin, Texas, 1965.

4. Green, C. C. Raphael, B. Research on intelligent question-answering systems. AFCRL-67-0370, Stanford Research Institute, Menlo Park, Calif., May, 1967.

5. Kaplan, R. M. The MIND system: a grammar-rule language. RM-6265/1-PR, The RAND Corporation, Santa Monica, Calif., April, 1970.

6. Kay, M. The MIND system: a powerful parser. (forthcoming).

7. _____, Martins, G. R. The MIND system: the morphological-analysis program. RM-6265/2-PR, The RAND Corp., Santa Monica, Calif., April, 1970.

8. _____, Su, S. Y. W. The MIND system: the structure of the semantic file. RM-6265/3-PR, The RAND Corp., Santa Monica, Calif., June, 1970.

9. Quillian, M. R. Semantic memory. Semantic Information Processing, Minsky, M. (Ed.), MIT Press, Cambridge, Mass., 1968, 227-270.

10. _____ The teachable language comprehender: a simulation program and theory of language. Comm. ACM 12, 8 (Aug., 1969), 459-476.

11. Quine, W. V. O. Notes on existence and necessity. J. Phil. 40 (1943) 113-127.

12. Schwarcz, R. M., Burger, J. F., Simmons, R. F. A deductive question-answerer for natural language inference. Comm. ACM 13, 3 (March, 1970), 167-183.

13. Shapiro, S. C. The list set generator: a construct for evaluating set expressions. Comm. ACM 13, 12 (Dec., 1970), 741-744.

14. _____ A data structure for semantic information, processing. Unpublished Ph.D. dissertation, University of Wisconsin, Madison, Wisconsin, 1971.

15. _____ and Woodmansee, G. H. A net structure based relational question answerer: description and examples. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, 325-345.

16. _____, _____, Krueger, M. W. A semantic associational memory net that learns and answers questions (SAMENLAQ). Technical Report #8, Computer Sciences Department, U. of Wisconsin, Madison, Wis., Jan., 1968.

17. Simmons, R. F., Burger, J. F. A semantic analyzer for English sentences. SP-2987, System Development Corporation, Santa Monica, Calif., Jan., 1968.

18. _____, _____, Schwarcz, R. M. A computational model of verbal understanding. SP-3132, System Development Corporation, Santa Monica, Calif., April, 1968.

19. Woods, W. A. Semantics for a question-answering system. Mathematical Linguistics and Automatic Translation Report No. NSF-19 to the National Science Foundation, The Aiken Computation Laboratory, Harvard University, Cambridge, Mass., September, 1967.