A DATA STRUCTURE FOR SEMANTIC INFORMATION PROCESSING

BY

STUART CHARLES SHAPIRO

A thesis submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN

1971

A DATA STRUCTURE FOR SEMANTIC INFORMATION PROCESSING

Stuart Charles Shapiro

Under the supervision of Associate Professor Larry E. Travis

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 dissertation describes a data structure and procedures for manipulating it that have been designed to meet the requirements outlined above. This system is intended to be used as the memory of a natural language question-answering machine 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.

Several topics discussed in this dissertation are relevant to wider areas of computer research and development, viz.: methods for the maintenance of list structures in virtual memory; algorithms for the very efficient, parallel evaluation of set expressions; measures for comparing which of several pieces of information, all relevant to

a given topic, are most specific to it; methods of recognizing "interesting theorems," i.e., facts derived via a chain of deductions that would be more valuable to save than to have to rederive. The use of the system to experiment with various semantic theories is demonstrated by examining several questions of current linguistic theory in the context of the system.

All the procedures for storing information in 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.

Acknowledgments

I sincerely appreciate the aid and guidance provided by Professor Larry E. Travis throughout my graduate career and during the research for and preparation of this dissertation. I also wish to thank Professors Leonard Uhr and Richard Venezky for their long term interest and the helpful discussions I have had with them. Professors J. Ben Rosen and Charles Davidson joined Professors Travis, Uhr and Venezky on the orals committee and I thank them for their interest and participation.

I am grateful for the support of The Rand Corporation and its Linguistics Group. I especially appreciate the help and insights of Martin Kay, head of that group. Gary Martins and Ronald Kaplan were also generous with their help and stimulating discussions.

A special thanks is due my colleagues and fellow graduate students, George Woodmansee and Phyllis Roney, for years of discussions, mutual teaching and moral support.

The research reported in this dissertation and the production of the dissertation were partially supported by a grant from the National Science Foundation (GP-7069) and partially by USAF Proj. RAND (project No. 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.

Table of Contents

List of Tables and Figures

Tables

Figures

1. Introduction

1.1 General Introduction

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. It has been designed to serve this latter role specifically within the MIND (Management of Information through Natural Discourse) System being developed at The Rand Corporation (see [29], [30], [31], and [32]).

1.2 Computer Understanding of Natural Language

An early impetus to research on machine understanding* of natural language was Bar-Hillel's argument that fully automated high quality translation could not be realized without the machine's having a knowledge of the world [5]. Later, the National Academy of Sciences' report, "Language and Machines" [45], recommended that support be withdrawn from machine translation projects in favor of more theoretical computational linguistics. This happened to a large extent and most groups then worked on syntactic analysis. Work on semantic analysis, i.e., extracting information from sentences for later use, and related studies at first appeared mainly out of artificial intelligence groups.

There are several ways one may demonstrate his understanding of a language, including: giving paraphrases, précis or translations into another language; answering questions based on material from that language; discussing the significance of or elaborating on material in the language. Computer programs may also use these methods to demonstrate their understanding of natural languages. Programs that give paraphrases or précis include Klein's automatic paraphraser [34] and Quillian's TLC [48]. No one has yet returned to Bar-Hillel's challenge and written a program to understand and translate. There are also no programs which, given a sentence or small text,

---

*True understanding requires much more than any computer program has yet been provided (see Kochen et al [37]). By building machines that understand natural language we mean building machines that, more and more completely, behavioristically appear to understand.

proceed to give a discourse on related topics although conversation programs such as ELIZA [71] and belief structure programs [1, 6, 13] represent starts in this direction. The vast majority of programs which are designed to understand natural language are question-answering programs. Many of these have been surveyed by Simmons [62;63]. Question-answering does seem to be the best context in which to experiment with computer understanding since the questioner can directly test whatever aspect of comprehension he is interested in.

Perhaps the most important criterion for understanding a language is the ability to relate the information contained in a sentence to knowledge previously acquired. This implies having some kind of memory structure in which the interrelationships of various pieces of knowledge are stored and into which new information may be fitted. Such structures are proposed for human memory by the cognitive psychologists [3, 46] and the structural semanticists [43]. They also exist in several computer programs including Semantic Memory [47], TLC [48], Protosynthex II and III [58;64;65], GRAIS [19], and SAMENLAQ [60;61].

The memory structures in these programs may be regarded as semantic, cognitive, or conceptual structures to the extent that they represent the relationships between the concepts stored in them and can use these relationships to make meaningful statements involving the concepts. It is important that these programs can make statements or answer questions based not only on the individual statements they

were previously told, but also on those interrelationships between concepts that were built up from separate sentences as information was incorporated into the structure. It is consistent with the theories of both cognitive psychology and structural semantics that the meanings of the terms stored in the memory are precisely the totality of the relationships they have with other terms in the memory.

The system to be described in this paper uses the question-answering paradigm to demonstrate its understanding. It stores data in the form of a net structure, providing a high degree of interrelationships between the stored concepts. It also can store general statements representing conceptual relations which it can then use to deduce information that is merely implied by the specific facts it has been told.

1.3 Information Retrievel

Information retrieval (IR) systems are generally used to retrieve a subset of the records of a file according to the values of certain of the fields of the records. They perform operations on the retrieved data, including sorting, counting, forming sums or averages of certain fields, formatting the data and producing reports. Systems that operate on documents or bibliographical data are used for producing literature review bibliographies and citation indices, providing current awareness services through selective dissemination of information, and retrieving documents or their citations as a response to specific reference service requests. Requests for information from IR systems are mainly in the form of boolean combinations of field values or descriptor terms, often with trained personnel designing the request from information supplied by the user. There are some systems that automatically analyze documents and/or requests, compiling descriptor terms for searching. These systems mainly use morphological analysis, thesaurus lookups, and a small amount of syntax, mostly statistics of coöccurrence, but major dependence is still on relatively simple word matching procedures. Current research on IR systems deals mostly with file organization, indexing methods, and measures of retrieval accuracy.*

*More extensive surveys of current systems and research and outlooks for the future may be found in [2], [14], [27] and [36].

A distinction is made between document retrieval and fact retrieval. If a researcher wants a list of all publications dealing with the determination of the diameter of the Earth, his request can be handled by a document retrieval system. If, however, he wants to know what the diameter of the Earth is, he requires a fact retrieval system. There is considerable overlap between document retrieval and fact retrieval since we may consider each sentence or piece of information stored in a fact retrieval system to be a document and the fact retrieval process to be the retrieval of the document(s) that can answer the question. On the other hand, "The following documents are relevant to your request: ..." is a fact that might be retrieved by a fact retrieval system. Fact retrieval thus includes document retrieval, although the converse is not true, since fact retrieval systems are expected to perform analysis and synthesis of the material stored in them. Fact retrieval systems in the research stage generally go under the name of question-answering systems, and are more concerned with syntactic and semantic analysis of the information stored in them as well as the requests (questions) made of them than are most IR systems.

MENTAL, the system described in this paper, is a question-answering system whose emphasis is on the semantic relationships of the stored information.

1.4 Deduction in Question-Answering Systems

Almost all question-answering programs have at least some ability to respond to questions with other than just the statements they previously have read in. In fact, question-answerers that just output the stored sentence(s) most likely to answer a question, such as Protosynthex I [66], should be classed as document retrieval programs rather than fact retrievers. The process of answering a question with a sentence derived from one or more input sentences but not actually the same as one of them is a process of inference. Whether the inference is inductive or deductive depends on the relation of the input sentences to the output sentence. Although several programs have been written to perform induction (see [69;28]), only a few are in the format of a question-answerer, e.g., Becker's [6]. Some question-answerers have been written to include modal deductions and even "non-logical" deductions (A⊃B,B ⊢ possibly A) (e.g., Colby et al. [13]), but most question-answerers stress strict logical deductions.

Several general methods may be noted in the ways deduction has been incorporated into question-answering systems. These may be divided into the three categories of (i) deduction while storing, (ii) deduction by executive routines, (iii) deduction by executive routines and stored rules. Any given question-answering system may exhibit deduction capabilities in several categories and subcategories.

Before discussing the categories, we will introduce the concept of a deduction rule. We will consider to be a deduction rule any statement, rule, or procedure which provides information as to what evidence is needed to deduce some statement(s). It is thus contrasted with a specific fact. Often, some mechanism is needed to interpret deduction rules. The mechanism uses a deduction rule as a recipe to produce specific facts from other specific facts. Some deduction rules, stated in English are: all whales are mammals; two things each equal to a third are equal to each other; one's male parent is his father; John is either at home or at his office.* Deduction rules may be used to provide a knowledge of terminology of a language, thus being similar to Carnap's meaning postulates [11]. Some deduction rules, along with specific facts, provide the knowledge of the world which Bar-Hillel [5] pointed out is necessary for translating natural languages and which Carnap [10] regards as the empirical evidence required to establish F-truth. "Deduction rule" as used in the present paper should not be confused with "rule of inference" as used in the literature of symbolic logic. A rule of inference is a rule for deducing a true statement from one or more true statements and is used to produce successive lines of a formal proof. A rule of inference is a

*Note how broad the concept of "deduction rule" is. The only statements that are not deduction rules are those that give specific facts, yet these are the only ones handled by ordinary IR systems.

deduction rule, but so are the general statements and theorems the rule of inference applies to and produces.

The category of deduction while storing has two quite different subcategories. The first, which migh be called immediate generation, is the method used in LISP-A [56] in which, at the time a deduction rule is input, all immediate implications of the rule are generated and stored. The rule is also stored and later information is tested against it with all new immediate consequences being generated and stored at the time the information is entered. This method is costly since facts are generated and stored that might never be asked for. The other method in this category, deduction by structure, gives a lot for relatively little cost and is used by almost all question-answering systems. It involves storing the information contained in input sentences in a data structure in such a way as to intimately relate it to the information in previous, relevant sentences, rather than storing each sentence as a separate entity. The way this works and its value may be seen easily in Lindsay's SAD SAM [41]. The sentences, "John is Sam's father" and "John is Bill's father" will cause a family unit to be created in which John is the father and Sam and Bill are children. From this structure the sentence, "Sam and Bill are siblings" may be generated as easily as if it had been an input sentence. Although it is a deduction from the previous two sentences, it requires no deductive processing of SAD SAM. The second sentence was stored as if its retrieval were all that would be required, but due to the

nature of the structure, the information in the third sentence appeared in memory as soon as the second sentence was stored. In a system that stores sentences as such separately, such as Fischer Black's [7] or Colby and Smith's [13], deductions of this type would require actual processing.

The second category, deduction by executive routines, also has two subcategories which may be called single search and double search. The single search method is in effect a version of the deduction by structure method in that complex retrieval languages and/or search routines are used with well-defined, predetermined structures for data storage. The searches resulting from a compilation or interpretation of the retrieval request find the basic data in the storage and do the necessary computations to answer the request. Prime examples of this method are the business-oriented information retrieval systems which provide languages to allow the users to specify search routines on record-oriented or hierarchical files. The Relational Data File with its INFEREX language [40] is also of this type as is Baseball [24]. Going further in this same direction are the systems that translate natural language questions into routines for searching predefined data structures, which routines are completely determined before searching begins, e.g., DEACON [15], Woods' [72] and Kellogg's [33]. Whereas, in the single search method each request is translated into a fixed search routine which is sure to find the answer if possible, in the double search

method the request is translated into a search routine which, if it fails to find the answer can be changed successively into other search routines based on the terms of the original routine and programmed deduction rules. Two classes of programs which use the double search method are those which have special programs for specific relations and those which have special programs for classes of relations. Examples of the former are SIR [52] and Protosynthex III [58;65] (in its use of the set inclusion relation). Examples of the latter are Elliott's GRAIS [19], in which there are programs for 32 different classes of relations, and Protosynthex III [58;65] which has programs for five different properties which a relation may be declared as having.

The final category of deduction methods, deduction by executive routines and stored rules, has subcategories that vary as to the amount of theorem proving power expended on manipulation of the deduction rules. This method may be considered a triple search method since it exhibits the two searches of the double search method as well as inserting between them a search for deduction rules, which are stored with the data rather than being embodied in programs. The first subcategory of this method consists of systems in which definitions of relations may be stored in the data structure. When answers to questions using these relations are not found explicitly in the data storage, the definitions are found and used to formulate additional searches.

Examples of these systems are Protosynthex III [58;65] in its provision of five operations for combining relations, SAMENLAQ [61] and SAMENLAQ II [60].

The second subcategory is those systems that can store and use general deduction rules in the form of implications. These use simple theorem proving techniques such as backward chaining: if the explicit information cannot be found, find a deduction rule whose consequent is a generalization of the information required, and search the memory for an appropriate instantiation of the antecedent. Examples of these systems are Fischer Black's program [7] and Colby and Smith's belief system [13]. The last group of systems in the third category are those that use full blown automatic theorem proving techniques on a data base whose specific facts and deduction rules serve as premises. Two examples of such systems are those of Green and Raphael [25] and of Darlington [16]. (Actually these systems use a refutation technique, Robinson's resolution principle [55], and thus do not have three separate searches but one deductive computation which produces the answer regardless of whether deductive processing was needed for the question or not.)

In terms of this discussion, MENTAL carries out deduction by structure and deduction by executive routines and stored rules. MENTAL fits within the second subcategory of this latter category, as very general deduction rules may be stored and the executive contains a general interpreter for using these rules by backward chaining. MENTAL differs

from other systems in this subcategory in that the structure of its memory makes finding relevant deduction rules a relatively quick, easy process.

1.5 Data Structures

For information retrieval and question-answering systems it is important that some structure be imposed on the data to facilitate searching and retrieval of interrelated data (although increased structure generally causes more costly updating). Data files that consist only of a collection of records, each record having several fields containing data, are limited in their possibilities of organization to the order in which the records are stored. They may be sorted on at most one field, which makes searching according to any other field very difficult. This difficulty may be partially avoided by representing the file several times, ordered on a different field each time. Actually, this only requires one master file with each subfile containing the ordered contents of one field plus the addresses of (pointers to) those records in the master file for which that field has that value. This is the rationale for the various forms of inverted files (see [18]).

In order to make the logical organization of the file independent of the physical organization more use is made of pointers. This not only allows additions and deletions to be performed without copying the entire file, but also allows organizations other than serial succession to be represented. Thus, linked lists allow logical succession to be divorced from physical succession; doubly linked lists (see [70], [18]) allow for traversal of the list in either direction; rings allow the entire list to be available after entry at any element; sublists allow for

hierarchical ordering; and attribute-value lists allow several different hierarchies by providing for labels for sublists.

Lists can be used to structure data as one would like to picture it, e.g., Lindsay [41] used them to build family trees. SIR [52] used attribute-value lists to represent collections of properties of individuals and Quillian [47] used the values of attribute-value lists as names of lists, thus getting an interconnected net structure. Reitman [53] also presents a structure wherein every element of a list is also the name of a list that describes that element.

Data structures for simulations and for graphical interactions typically consist of record blocks each consisting of several fields whose contents are data and several fields whose contents are pointers to other record blocks. The significance of the data and pointers is determined by the order they appear in the record block rather than by labels appearing in the record.

Associative memories also consist of blocks or cells divided into fields containing data or pointers, the fields having known locations within the cells. Particular cells are located by paths of pointers from known cells by giving a sequence of fields, pointers being followed from successive cells according to that sequence (e.g., in L^6 [35], ABC refers to the C field of a block pointed to by the B field of a block pointed to by the base register A). In some cases cells are located by the contents of their data

fields via some computation on the required value and the field that value is supposed to be in (e.g., LEAP [20] stores data triples in three separate files, organized by hash coding the values of the different fields). Associative memories are used for fact retrieval by storing relational statements in the cells, using one field for the relation and the others for the arguments (e.g., see [20]). An argument may be an entire statement by having the appropriate field contain a pointer to another cell.

Question-answering systems that use a structured data base and are supposed to be general rather than specifically designed for one subject-matter field generally make use of associative memories of list structures including rings and sublists and/or attribute-value lists. They generally store information in the form of relational statements where the order of fields determines what part each term takes in the statement (i.e., relation or 1st argument or ...). Most of these systems use a triplet structure, representing all information as nests of binary relations.

MENS, the data structure described in this paper, is an associative memory in which the number of fields within a cell is determined by how that cell is related to others and may vary during the life of the cell. Because of this and because different cells may have different fields, the fields are labeled rather than being in particular locations within the cell. Lists of cell names are used when some field of a cell points to more than one other cell. Since a cell is made up of labeled fields with each field

containing a pointer or list of pointers to other cells, it is like an attribute-value list. Some cells are like base registers in that they are locatable via a symbol table from a symbolic name. All cells are also locatable from other cells via paths of field labels.

2. The Basic Structure -- Explicit Storage and Retrieval

2.1 Introduction

In this chapter we describe the MENS structure and the basic MENTAL system. In Section 2.2 the motivating factors behind MENS and MENTAL are presented and it is explained how they led to the data structure and the system to be described. Sections 2.3 and 2.4 discuss the structure from the machine implementation point of view. Section 2.5 discusses the MENS structure, storage into it and explicit retrieval from it from a more abstract view -- one which a user rather than an implementor would be interested in. It also presents the input language used when MENTAL serves as an interactive system detached from a natural language interface. Section 2.6 presents the list set generator, a generalization of the method used in MENTAL for taking the union and intersection of arbitrary numbers of ordered lists. The system as presented in this chapter has been programmed in PL/1 and is running on an IBM System/360 with an interactive terminal at The Rand Corporation.

2.2 Basic Concepts of the Structure

There were several motivating factors for the MENS structure:

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: The 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.

The first four factors listed above were criteria for the original MENS structure [59;60]. The last two necessitated the expansion into the present MENS structure, which is to be described in detail in this paper. In this section, we will describe how the 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. 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 nonconceptual 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 labeled 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 labeled 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 retrievable 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.

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 2.5. 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 able to be 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.

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. In the original version of MENS, there were three pairs of links -- left, middle and right* and their reverse links. A relational statement, a R b, was represented in the structure by an item block, s, which had a left pointer to a, a middle pointer to R and a right pointer to b. Unary relations (one place properties) were represented by a block one of whose forward pointers was not used. The n-ary relations, for

*The three links may also be considered to be agent, verb and object or object, attribute and value or first argument, relation and second argument.

n greater than two, would have the right pointer going to an item block which served only to point to some more arguments. If n were greater than four, at least two levels of such items were used. These items had no conceptual significance and made searching less efficient, which is why direct representation of n-ary relations was adopted. It requires that any item 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.

2.3 Implementation of Item Blocks and Disk Lists

An item block is a consecutive block of records each consisting of three fields. The first field holds the internal name of a link related to the external name of a system-relation through a symbol table. One bit is used to distinguish the forward link of a link pair from its reverse link (0 for forward, 1 for reverse). This is done so that, given the internal name of any system-relation, the internal name of its converse is derivable by simply complementing one bit. Which label of a pair is the forward one and which the reverse is, of course, arbitrary and fixed by the user of the system for his own convenience.

The second field of each record in an item block is a single bit which is set according to whether the link in the first field is single or multiple (0 for single, 1 for multiple). Whether a particular link is single or multiple is fixed by the user and any of the four possible combinations may hold for a (forward and reverse) link pair, depending on whether the relation the user wishes the link pair to represent is 1-1, 1-many, many-1 or many-many.

The contents of the third field of a record depends on the contents of the second field. If the second field is 0, the third field contains the internal name of an item block. This name consists of the number of the disk track on which the item block is located followed by the offset giving the location of the first record of the item block within the track. The item thus pointed to is in the

relation given by the link in the first field of the record to the item in the block of which the record is located.*

If the second field is 1, the link in the first field is multiple and instead of containing the internal name of an item block, the third field contains the name of a list which contains the internal names of all the items in the given system-relation to the item block in which the record is located. The name of such a list is, like the internal name of an item block, the location of the list on the disk.

When an item block is first created, room for a certain number of records is set aside for it. It may subsequently happen that all these records are filled and an attempt is made to attach a new link to the item block. When this happens, another block of records is set aside for this item block and made a continuation of the item block. The new record is put in this new block and the location of the continuation block is recorded in a continuation directory for the original block. If at all possible, the continuation block is put in the same disk track as the original block so that, abiding by the motivating factor of single file of the previous section, only one disk access will be necessary to get all the information in the item

*In the actual implementation of item blocks and lists on disk written at The Rand Corporation by S. Y. Su, the internal name of an item gives the location in a directory of the actual address of the item block. There are other places where the implementation differs from the description given in this section, but they are described in [28] rather than in this dissertation.

block. This often will be possible since room is set on each track for continuations of blocks that are already on that track.

The list which holds the pointers for a multiple link is also begun on the same track as the item block in which its name is the third field of some record. Again, an attempt is made to keep the list on one track. If this is successful, then the requirements of single file are eminently satisfied.

This method of storing the multiple pointer lists may be compared with a ring structure method, which is an alternative that readily presents itself. In the ring method, the third field of the record containing a multiple pointer would point to a field of one of the item blocks being pointed to by the multiple link. That field would contain a pointer to another of the item blocks being pointed to, etc., until the last item block on the ring would point back to the item block containing the multiple pointer. It might seem that the ring structure saves space since only the pointer to the next item block must be stored in each field, whereas in the list method, each element must contain an item name plus a pointer to the next list element. However, there must be some mechanism to enable the system to determine in which item block the ring field is located and as we shall see, in the list method used, very few pointers to list elements are needed.

The major advantage of the list system, though, is that it meets the single file requirements. If we wish to know

which items are connected to a given item by a certain multiple link originating in that item, the list method requires one disk access, whereas the ring method requires one disk access for each item on the ring.

In order to reduce the space requirements of storing the multiple pointer lists, a compromise was made between arrays and lists. List elements are stored in blocks of consecutive fields, each field holding one item name, with one field containing a pointer to the next list block. Thus the lists are similar to Hansen's compact lists [26]. The lists are ordered according to the numerical values of the item names for reasons to be explained in the next section, so there are two different problems which arise when trying to minimize the number of disk accesses required to use the file. First, a list may become spread out over many tracks as it grows. Second, as elements enter in the middle of lists, the lists may jump back and forth between tracks. The first problem would cause many different tracks to be read in the course of one list read. The second would cause several tracks to be read into core and written out to make room for others, only to be read back in again perhaps only one list element later. These two problems call for the solutions: 1) as much as possible, keep a list within one track; 2) when a list must extend to another track because there is simply no more room on the first, extend the end rather than the middle of the list and use for the new track one with enough room that there is little chance that the list will soon have to extend to a third

track. Besides this, we will, whenever possible and reasonable, move the extension of a list back into the original track.

The solution to the first problem involves keeping a cushion of space on each track which will be used only for extensions of lists which are already on that track. When a track has only its cushion left, no new list will be started on it. Furthermore, if a track does run out of space, its extension will be started on a track with more room than just its cushion. In this way, we try to insure that a list will reside on as few different tracks as possible.

The solution to the second problem assures us that if a list does extend to another track, once we proceed to the next track in reading down the list, we will not have to return to a previous track. This is done in the following way. When a pointer is to be added to a list, it is first determined where on the list the new pointer should go. If there is room on that block, the other pointers on the block are adjusted appropriately and the new pointer is inserted. If there is no room on that block, but there is on the succeeding or previous block (if such exists on the same track), then that space is used with appropriate adjustment of pointers. Thus, if we can find empty space on the list nearby, we use it, but if we can't find it that close, we do not look further since this might require moving pointers up or down in and across list blocks, thus ignoring one of the most useful properties of linked lists. Instead, we

get a new list block to use. The space thus left scattered throughout the list will be used when a pointer is to be stored nearby, or else may eventually be gathered together into blocks by the packing garbage collector described below. So, if the track we are on has available list blocks, we simply get a new block and link it into the list after the block on which the new pointer is to be inserted. If there are no empty blocks, then it would seem that the list must extend over onto a new track. (Sometimes this will not be necessary. See below.) If, however, the list already runs over to a second track we do the following. Get a new block on that next track (if there is room -- this process is recursive), move the pointers that are on the last block of the list on the first track, put them onto the new block on the next track, and use the block just vacated (appropriately relinked) to store the new pointer. In this way we help insure that when a list runs over to a new track, the entire tail runs over -- it does not later come back. If, of course, the new pointer was to go on the last block of the list on the first track, the new block on the next track is used directly, as if it were on the first track, and an empty block is left on the first track for later insertions. If the list does not already extend to a new track, it is worth some extra effort to try to keep such extension from occurring. Therefore an attempt will be made in this case to "pack" the lists on that particular track. This is done by reading down each list on the track and keeping note of empty spaces on list blocks. Every time enough spaces are

found on any list to form a block, the pointers which occupy the portion of the list where the spaces were found are moved so that the spaces are brought to one list block. That block is then released to be empty space. In this way one or more new blocks become available for use on the track and the list need not extend onto another track. If this packing cannot be done, because no list on the track has enough empty pointer spaces, we have no choice but to get another track and extend the list onto it. Of course, if at least one empty pointer space was found on the list we are interested in, we can use it instead of extending to a new track.

After a list has extended onto several tracks it is possible that space becomes available on one of the early tracks. This would allow part of the list to be moved back onto that track, possibly reducing the total number of tracks the list uses. This is done as follows. If, while reading a list, we must go to a new track, we notice if the earlier track has more available space than its cushion. If so we will fill the extra space with the top blocks of the list on the next track. Possibly, by this time we will have moved the entire tail of the list back. If this has not occurred, the cushion will be used only if it will hold the entire portion of the list that is on the next track, thus eliminating one disk access when reading that list in the future.

2.4 Core List Processing System

To provide for temporary storage and working lists, an in-core list processing system has been written as a part of the MENTAL system. The core list system is patterned after SLIP [70], but the lists are singly linked, the last element of a list has a null pointer rather than pointing to the header, and the header, which contains pointers to the first and last elements of the list and has a field containing the length of the list, does not contain a reference count. Because of the lack of a reference count and because of the way garbage collection is done, a list can be a sublist of at most one other list. This was done to keep the list processing system as simple as possible. List readers, which require the space of two normal list elements, can be elements of lists, thus allowing list set generators (Section 2.6) to be built. One of the most important properties of core lists is that disk lists (see Section 2.3), although formatted differently from core lists, may be sublists of core lists, and readers can read both core lists and disk lists.

Core lists are contained in a list space structure with the following divisions: a vector called HEAD of two elements each 15 bits long; a vector called TAIL of two elements each 15 bits long; a vector called ELT of 2000 elements each 6 characters (48 bits) long. HEAD(1) and TAIL(1) point to the beginning and end, respectively, of the available space list of single elements, while HEAD(2) and TAIL(2) point to the available space list of double

elements (for list readers). Single and double elements are distinguishable by the contents of an ID field (see below). Initially, all 2000 elements are tied together into the available space list of double elements, making for 1000 elements. The subscript of the ELT array which identifies the location of a single element or of the first element of a double element will hereafter be referred to as the index of that single or double element. A double element will always consist of two consecutive members of the ELT array. Every list element contains a 2 bit ID field with the following values:

'00' -- a single element which contains data.

'01' -- a single element which contains the index of a core list header or the address of a disk list.

'10' -- a single element which contains the indices of the first and last elements of a core list and the length of that list.

'11' -- a double element which serves as a reader of a core or disk list.

These elements are referred to as a data element, a list name, a header, and a reader respectively. The fields of each kind of element are shown in Table 1 and described below.

A data element contains a 2 bit ID field, a 15 bit NEXT field and a 31 bit DATA field. The NEXT field contains the index of the next element in the same core list as the element itself. This field contains all zeroes for the last element in a list.

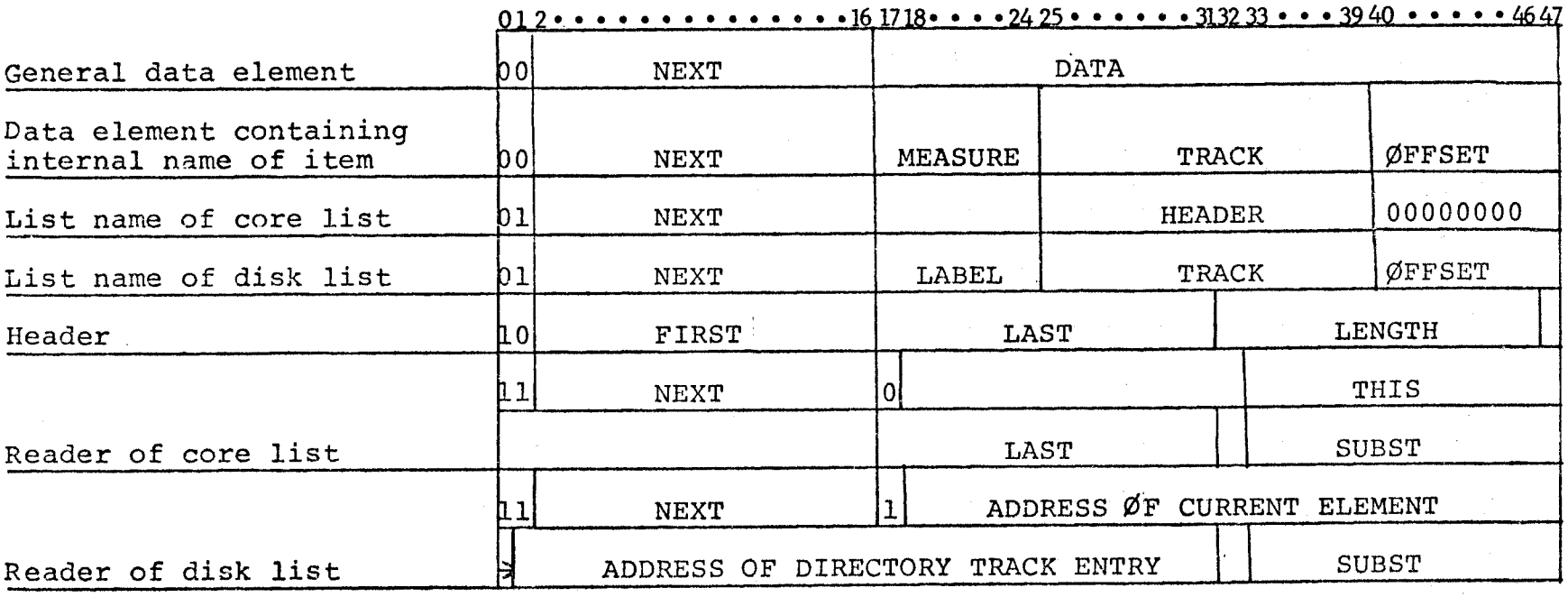


TABLE 1 - Use of fields in core list elements.

(Blank fields are not used.)

When a data element contains the internal name of an item, the DATA field is divided into an 8 bit MEASURE field, a 15 bit TRACK field and an 8 bit OFFSET field. (See Section 2.3 for the explanation of the internal names of items and disk lists.)

A list name has the following structure: a 2 bit ID field; a 15 bit NEXT field; an 8 bit LABEL field; a 15 HEADER field; an 8 bit OFFSET field. When this element is the name of a core list, the HEADER field contains the index of the header of the list and the OFFSET and LABEL fields contain all zeroes. When it is the name of a disk list, the HEADER and OFFSET fields contain the track and offset, respectively, of the item from which the list emanates as a multiple pointer list, and the LABEL field contains the internal name of the label identifying the multiple pointer.

A header has the following structure: a 2 bit ID field; a 15 bit FIRST field; a 15 bit LAST field; a 15 bit LENGTH field; a 1 bit MARK field. The FIRST field contains the index of the first element of the list or zeroes if the list is empty. The LAST field contains the index of the last element of the list or the index of the header itself if the list is empty. The LENGTH field contains the current length of the list. The MARK field is not used at present.

A reader has: a 2 bit ID field; a 15 bit NEXT field; a 1 bit TYPE field; a 62 bit RDATA field; an unused bit; a 15 bit SUBST field. The TYPE bit is '0' if the reader is reading a core list and '1' if it is reading a disk list.

The SUBST field contains the index of the header of a list structure which contains substitutions. SUBST is used when the reader is being used as a primitive list set generator to evaluate list sets which are lists of instantiation specifications. In a core list reader RDATA is structured as follows: 15 unused bits; a 15 bit THIS field; 17 unused bits; a 15 bit LAST field. The THIS field contains the index of the element currently being pointed at by the reader. The LAST field contains the index of the element just preceding the one pointed to by the contents of the THIS field.

When the reader is reading a disk list, the first 31 bits of RDATA are used to hold the disk address of the element being pointed at and the last 31 bits are used to point to the entry on the directory track for the list segment the element is in.

Core lists may be catagorized into four classes such that any one list must belong to exactly one of them. They are:

1. Reader lists: All the elements are readers.
2. General lists: Elements are data elements and/or list names (of core or disk lists) in any order.
3. Data lists (list sets): Elements are all data elements whose data are internal names of items. The elements are ordered on the numerical values of their data fields, largest first.

4. Lists of instantiation specifications: Data lists or lists of data lists, with some data elements (or list names) followed by substitution lists.

A substitution list contains a substitution (see Section 2.5), with odd elements being the variables of the substitution components and even elements being the terms of the components (either a data element or a sublist). The data element or list name which is followed by a substitution list has its NEXT field pointing directly to the header of the substitution list. The last element of the substitution list has its NEXT field pointing to the next element of the data list (or list of data lists). This has been done to distinguish substitution lists from regular sublists, while allowing the null substitution to be simply the absence of any substitution list, and lists of instantiation specifications to be handled by the same procedures no matter how many of their substitutions are non-null.

2.5 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 speclist, which describes the item by describing the paths in the net that lead away from the item. These paths may be quite complicated, but every one must end at an item that is known both to the user and to the system, and 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 angle 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 → $\left\{\begin{array}{l}\langle \text{any integer i, } 0 \leq i \leq 2^{31} \rangle \\ \# \end{array}\right\}$

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 (see section 2.2 for the distinction between system relations and item relations). 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 an 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. The three fields are: the specification measure of the item (see Section 4.5); the track number of the item; the offset within the track of the item. Examples of cnames are:

JOHN

UNITED_STATES

HAS_SENSE_1

241/00010+023

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 in 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 associations. It is important to distinguish between variable items and vnames. Variable items (see Chapter 3) 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

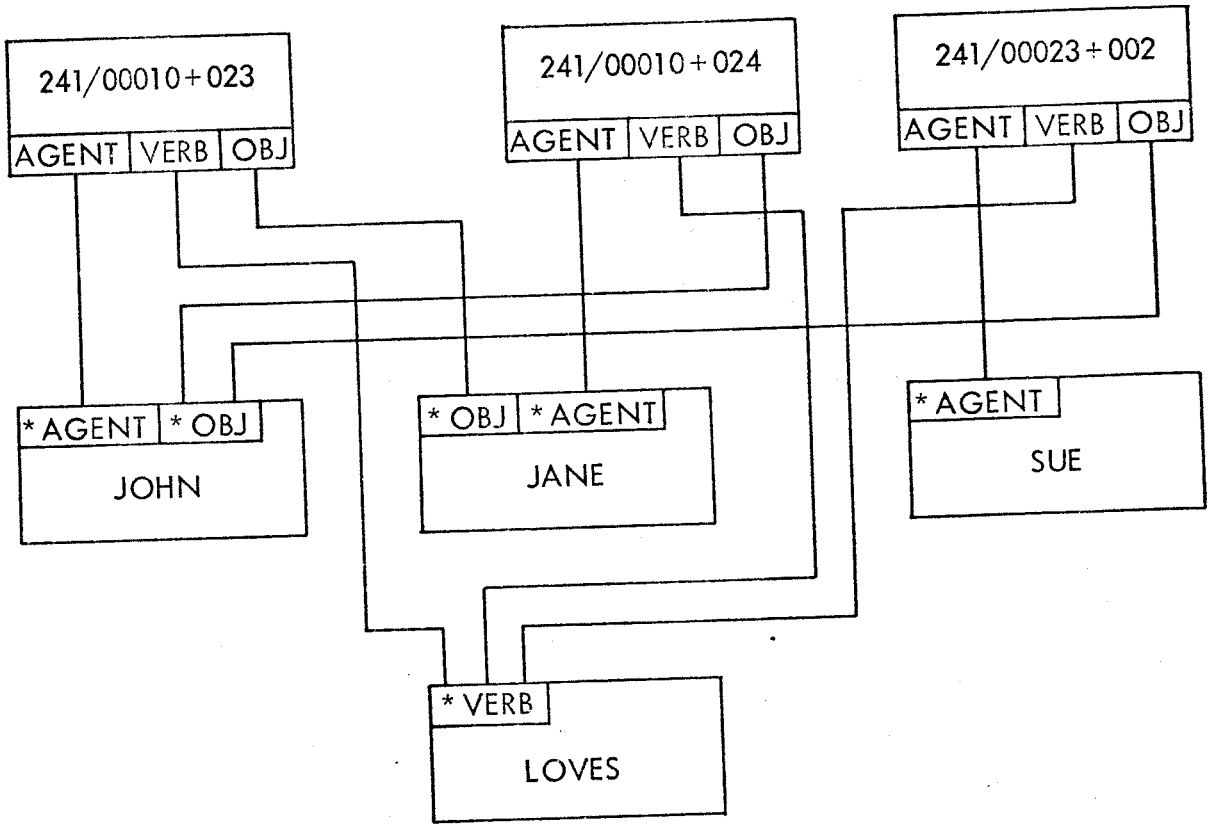


Fig.1 — A MENS substructure, described in the text

item that will not have an external name. This is precisely the purpose of using a vname in this way. 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 value of that spec is a list of items which are removed from the value of the speclist. This allows a retrieval request of the form: "Tell me all items described by $spec_1$ 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:WRITE))-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 in 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 representing

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 core 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 %name 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 whose name is given. What has thus been described is a substructure of the net structure, and at the center of the substructure (or, we may say, at the head of the substructure) is the item described by the spec. It may be that the description fits more than one substructure of the net. In this case, the value of the spec is a list of all items that are heads of the substructure so described. If no substructure 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 existence 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 chapter 3). If the number of items found exceeds the second num, there is a semantic ambiguity that must be cleared up (see section 4.3). The character "#" is used to represent the largest integer than 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 being 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 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 whever 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 represent 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 the *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 *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 explained in section 2.6.

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 section 2.2. 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 purpose of such a vname is to specify that some unknown block is reachable by several different paths from the head of the sub-structure 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. To understand this more clearly, a detailed analysis of substitutions and their use in identifying substructure follows. (Where possible, the terminology of Reynolds [54] based on Robinson [55] is used or adapted.)

A general statement is a spec within which are one or more vnames.

The range r(G) of a general statement G is the set of vnames that appear in G.

A substitution component is an expression of the form V/T, where V is a vname and T is an item name or a list of item names no two of which are the same. The item(s) named in T may be constant items or variable items. V and T are called the variable and the term of V/T respectively.

A substitution is a finite list $(V_1/T_1 \ldots V_n/T_n)$ of substitution components no two of which have the same variable.

The domain d(s) of a substitution s is the set of variables of its components.

An instantiation specification of a general statement G is an expression of the form S:s where S is an item name, s is a substitution, d(s) = r(G), and if for every V/T in s, every occurrence of V in G is replaced by T (or any of the item names in T), the result will describe S. That is, S is the head of a substructure described by the spec G, and s records the association between vnames in G and items in the substructure headed by S. Note that the notation S:s

does not signify the transformation of S using the substitution s. For our purposes, we will be given G, and will discover S in the memory net. Simultaneously with discovering S we will build up s. The instantiation specification S:s is used for recording the fact that s is the substitution built during the discovery of the substructure headed by S.

A list of instantiation specifications is a list of the form $(S_1:s_1 \ldots S_n:s_n)$ or $(S_1 \ldots S_n):s$ where for $1 \leq i \leq n$, $S_i:s_1$ is an instantiation specification (IS) of some general statement; $(S_1 \ldots S_n):s$ is equivalent to $(S_1:s \ldots S_n:s)$.

We can now see that the value of a spec in the case where the spec is a general statement G is a list of instantiation specifications (LIS) such that every IS in the LIS is an IS of G. We will now see how the union and intersection operations necessary for the evaluation of specs are to be modified to allow for the union and intersection of LISs. Since in evaluating specs it will be necessary to intersect the values of specs that contained vnames with values of specs which did not contain vnames, we now define the null substitution to be a substitution with no substitution components, and we will consider the value of a spec that did not contain any vname to be an LIS all of whose substitutions are null. We will first define the u-merge and i-merge* of substitutions and then show how these operations are used in evaluating general statements.

*The "u" and "i" are from "union" and "intersection" respectively since the u-merge operation will be used when unioning LISs and the i-merge will be used when intersecting them.

The u-merge of substitutions $\underline{s}_1$ and $\underline{s}_2$ is the substitution $\underline{s}_3$ which contains a substitution component V/T for every V in $\underline{s}_1$ or $\underline{s}_2$ or both, with the list T containing every term in a component with V as its variable in $\underline{s}_1$ and every term in a component with V as its variable in $\underline{s}_2$. Symbolically:

$$\underline{s}_3 = \{V/T \mid \overline{\langle} \exists T_1 (V/T_1 \varepsilon \underline{s}_1) \vee \exists T_2 (V/T_2 \varepsilon \underline{s}_2) \overline{\rangle} \;\&\; \overline{\langle} \forall t (t \varepsilon T <=> \exists T_1 (V/T_1 \varepsilon \underline{s}_1 \;\&\; t \varepsilon T_1) \vee \exists T_2 (V/T_2 \varepsilon \underline{s}_2 \;\&\; t \varepsilon T_2) \overline{\rangle} \}$$

The i-merge substitutions $\underline{s}_1$ and $\underline{s}_2$ is the substitution $\underline{s}_3$ such that $\underline{s}_3$ contains a substitution component V/T if V/T is a substitution component of one of the substitutions $\underline{s}_1$ or $\underline{s}_2$ and there is no substitution component with V as its variable in the other substitution and $\underline{s}_3$ contains a substitution component $V/(T_1 \cap T_2)$ if V/T_1 is in $\underline{s}_1$ and V/T_2 is in $\underline{s}_2$ and $T_1 \cap T_2$ is non-null. Symbolically:

$$\begin{aligned} \underline{s}_3 = \{V/T \mid & \overline{\langle} V/T \varepsilon \underline{s}_1 \;\&\; \neg \exists T_2 (V/T_2 \varepsilon \underline{s}_2) \overline{\rangle} \;\vee \\ & \overline{\langle} V/T \varepsilon \underline{s}_2 \;\&\; \neg \exists T_1 (V/T_1 \varepsilon \underline{s}_1) \overline{\rangle} \;\vee \\ & \overline{\langle} \exists T_1 \exists T_2 (V/T_1 \varepsilon \underline{s}_1 \;\&\; V/T_2 \varepsilon \underline{s}_2 \;\& \\ & \qquad \forall t (t \varepsilon T \text{ F<=>} t \varepsilon T_1 \& t \varepsilon T_2) \;\& \\ & \qquad \exists t (t \varepsilon T_1 \& t \varepsilon T_2)) \overline{\rangle} \} \end{aligned}$$

Let us now consider the unioning of LISs. When will this be done? When a speclist G is evaluated, the result will be of the form $(S_1:\underline{s}_1 \ldots S_n:\underline{s}_n)$. If this speclist appears in a spec preceded by "L:", the value of the speclist will be changed to

(1) $\quad ((S_{11} \ldots S_{1m_1}):\underline{s}_1 \ldots (S_{n1} \ldots S_{nm_n}):\underline{s}_n)$

where $(S_{i1} \cdots S_{im_i})$ is a list of all items pointed to by the converse L pointer from S_i. This is clearly justified since if $S_i:\underline{s}_i$ is an IS of G, $S_{ij}:s_i$ is an IS of the general statement (L:G). It is the various LISs of (1) that are to be unioned.

Now, if L is a single pointer, S_{ij} cannot be the same item as S_{kl}, for $i \neq k$ and any j and l, because it would be impossible for any item to have an L pointer to both S_i and S_k. In this case, the unioning would simply require the distributing of the substitutions over their LISs and the reordering of the ISs into a single LIS. However, if L is a multiple pointer, it would be possible to have $S_{ij} = S_{kl}$. Let us assume S is such an item name, and it appears in (1) with the substitutions $\underline{s}_{i_1} \cdots \underline{s}_{i_n}$. How is S to appear in the unioned LIS? Clearly $d(\underline{s}_{i_1}) = \ldots = d(\underline{s}_{i_n})$ since $d(\underline{s}_1) = \ldots = d(\underline{s}_n) = r(G)$. However, it might not be true that $\underline{s}_{i_1} = \ldots = \underline{s}_{i_n}$ since some path of the substructure from the head to the item which is substituting for a vname in G may include a multiple pointer and one substitution may describe the substructure by following one pointer of the multiple pointer while another describes an alternate path. Specifically S has an L pointer to each item $S_{i_1} \cdots S_{i_n}$ and they are distinct items so there is no reason to suppose that their substitutions are equal. We will now show that the substitution that should appear with S in the final LIS is the one formed by taking the u-merge of all the substitutions $\underline{s}_{i_1} \cdots \underline{s}_{i_n}$.

Assume that $S:(V_1/T_1 \cdots V_n/T_n)$ and $S:(V_1/I_1 \cdots V_n/I_n)$ are ISs of the same general statement G with T_i and I_i being single item names, $1 \leq i \leq n$. It will then be true that $S:(V_1/J_1 \cdots V_n/J_n)$, where for $1 \leq i \leq n$, J_i is either T_i or I_i, is also an IS of G. The last IS states that if the paths defined by the placement of V_i in G are followed from S, J_i will be reached. If J_i is T_i, this is exactly what is stated by the first IS and if J_i is I_i, this is what is stated by the second IS. Thus, by the definition of IS, $S:(V_1/(T_1\ I_1) \cdots V_n/(T_n\ I_n))$ is an IS of G and the substitution in this case is the u-merge of the substitutions in the two given cases. This argument obviously extends to ISs whose substitutions contain terms that are lists of names and shows what we set out to show, viz.: the union of LISs, where every member IS is an IS of the same general statement, is the LIS consisting of the set of ISs $S:\underline{s}$ such that S appears in one or more operand LISs, and $\underline{s}$ is the u-merge of all substitutions $\underline{s}_i$ such that $S:\underline{s}_i$ is in an operand LIS.

If G_1 and G_2 are general statements and we are evaluating the spec $(\ldots L_1:G_1, L_2:G_2 \ldots)$, then after the LISs for the general statements $(L_1:G_1)$ and $(L_2:G_2)$ are found, we will want to intersect them. That is, we will be intersecting the LISs $(S_{11}:\underline{s}_{11} \cdots S_{1n}:\underline{s}_{1n})$ and $(S_{21}:\underline{s}_{21} \cdots S_{sm}:\underline{s}_{2m})$ to find an item S that is the head of the substructure described by $(L_1:G_1)$ as well as the substructure described by $(L_2:G_2)$. Say S_{1i} is the same

item as S_{2j}. In general it will be the case that $d(\underline{s}_{1i}) \neq d(\underline{s}_{2j})$. Let us see how we want to construct the substitution $\underline{s}$ to appear with S_{1i} in the intersected LIS. If V/T is a component of $\underline{s}_{1i}$ but is not a component of $\underline{s}_{2j}$, we want V/T to be a component of $\underline{s}$ since in the substructure headed by S_{1i} and described by $(L_1:G_1,L_2:G_2)$ any item in T is in the position defined by V. Similarly, if V/T is in $\underline{s}_{2j}$ but not in $\underline{s}_{1i}$. If V/T_1 is in $\underline{s}_{1i}$ and V/T_2 is in $\underline{s}_{2j}$, but T_1 and T_2 have no item in common, then we do not want S_{1i} in the intersected LIS at all because there is no item which can replace all occurrences of V in $(L_1:G_1,L_{2:G_2})$ resulting in a description of a substructure headed by S_{1i}. If, however, T_1 and T_2 have some items in common, then we want a component in $\underline{s}$ with V as its variable and the intersection of T_1 and T_2 as its term. Thus, in the case where S_{1i} is placed in the intersected LIS, the substitution $\underline{s}$ is the i-merge of the substitutions $\underline{s}_{1i}$ and $\underline{s}_{2j}$. Let us say that two substitutions are compatible if for all components V_1/T_1 of the one and V_2/T_2 of the other, if $V_1 = V_2$ then T_1 and T_2 have at least one item in common. Let us extend this definition and say that several substitutions are compatible if whenever there is a set of substitution components $V/T_1 \ldots V/T_n$ with each V/T_i in a different one of the substitutions, there is at least one item common to all the terms T_i, $1=i=n$. Now we can define the intersection of LISs, which we will call the compatible intersection of LISs, as that LIS consisting of

the set of ISs S:$\underline{s}$ such that there is an IS S:$\underline{s}_i$ in each of the LISs being intersected and all such $\underline{s}_i$'s are compatible and $\underline{s}$ is the i-merge of the $\underline{s}_i$'s.

2.6 The List Set Generator

2.6.1 The General LSG

As is obvious from the discussion in the preceding section, intersecting and unioning arbitrary numbers of sets of item names is performed frequently when retrieving information from MENS. It is therefore necessary to have a very efficient method for doing this. This section describes a generalization of the method which was found, which generalization allows for the very efficient evaluation of set expressions of arbitrary length and complexity. The techniques described below would be useful in language systems that have a set data type and in systems for manipulating ordered files as well as in associative data systems.

Unordered sets may be represented as lists which do not contain duplicate elements. The set operations will be performed more efficiently if the lists are ordered on some internal code (see, for example /~20_/). The set operations difference (relative complement), union and intersection, which could be performed only very inefficiently on unordered lists representing sets can be done efficiently on these ordered lists. For example, to intersect two unordered lists takes an amount of time proportional to the product of their lengths while to intersect two ordered lists takes an amount of time proportional to the sum of lengths. When intersecting more than two lists, even more time could be saved by reading all the lists in parallel rather than intersecting them by pairs. If three lists were to be intersected of lengths m, n and r and the first two had s elements

in common, intersecting them two at a time as unordered lists would take an amount of time directly related to mn+sr, the time to intersect them as ordered lists two at a time would be directly related to m+n+s+r, but the time to intersect them by comparing all three at once would be directly related to m+n+r. The same results would hold for the other set operations.

In this section we define a generalization of the list reader (see Weizenbaum's reader [70] and Knowlton's "bug" [35]) which, as it is incremented, produces the new set determined by set operations on given sets. The algorithms for incrementing the generalized reader embody the efficient parallel methods for performing the set operations on ordered lists.

We first introduce some basic terminology.

D1. A <u>list set</u> is an ordered, finite list no two of whose elements are equal.

The ordering relation used in list sets is immaterial. In fact, different orderings may be used on different lists and any equivalence relation may be used for equating elements of different lists. The restriction is that if two elements are equivalent, then no element that appears after one of them on some list set shall be equivalent to any element that appears on any list set before the other. If we let $\equiv$ be the equivalence relation, G(x,y,z) be a predicate that is true if and only if x and y are elements on the list set z with x appearing after y on z, e_1, ..., e_4 be variables over elements and L_1,L_2 be variables over list sets, this restriction may be expressed as:

$$\forall e_1 \forall e_2 \, \lceil (e_1 \equiv e_2) \Rightarrow \neg \exists e_3 \exists e_4 \exists L_1 \exists L_2 (e_3 \equiv e_4 \; \& \; \lceil (G(e_3, e_1, L_1) \& G(e_2, e_4, L_2)) \vee (G(e_1, e_3, L_1) \; \& \; G(e_4, e_2, L_2)) \rfloor) \rfloor.$$

This restriction, of course, induces a common ordering relation on all elements of all lists in any operation, but this might not be one that is easily applied directly to some of the sets in question.

In any implementation of these algorithms, it would be possible to represent ordered sets by having the user provide a function which, given two elements, returns one of three codes depending on whether the first element is greater than, equivalent to, or less than the second and using this function whenever two elements are to be compared. It would also be possible to use these algorithms on ordered attribute-value lists (or any list where the ordering is on every $n^{\underline{th}}$ element with the $(n+1)^{\underline{th}}$ through the $(2n-1)^{\underline{th}}$ elements always following the $n^{\underline{th}}$). For the purposes of this discussion, we will assume that all lists are ordered on an internal numeric code, smallest number first, and we will use identity as the equivalence relation.

Since, in the algorithms given below, a list is often searched for the smallest element equal to or greater than a given element, even more speed can be achieved if the lists are organized hierarchically. By this is meant that the list is divided into sublists each containing n elements, for some n, and a higher level list is used each of whose elements points to the beginning of a sublist and either also points to the end of the sublist or contains the value

of the last element on the sublist. This higher list can then be divided into sublists similarly. Then, when a given element is to be found, a search is done from higher levels to lower levels as the proper sublists are located. There are no changes required in the algorithms given below to accommodate normal ordered lists or hierarchical lists since the only changes needed are in the design of the reader and the routines to manipulate the reader.

A reader, as used in this discussion, may contain only a pointer to a list element or additional information as well. The essential requirements are that the reader identify a unique element of some list (which we will refer to as the element currently pointed at by the reader), and that it be possible to retrieve the datum of that element, to increment the reader so that it points to the next element in the list, and to recognize when the element it is pointing at is the last in the list set.

We can consider a reader as a generator of the set represented by the list it reads. We will define three other list set generators. A difference list set generator is used to generate a set which is the difference between the sets generated by two list set generators. A union list set generator is used to generate a set which is the union of the sets generated by a number of list set generators. An intersect list set generator is used to generate a set which is the intersection of the sets generated by a number of list set generators. Figures 2-5 and the sample problem in Appendix A demonstrate the use of these generators. The algorithms used are given below.

D2. A list set generator (LSG) is defined recursively as follows:

1a) A primitive LSG (PLSG) is a reader.

1b) A PLSG is an LSG

2a) A difference LSG (DLSG) is an ordered pair of LSG's

2b) A DLSG is an LSG.

3a) A union LSG (ULSG) is an ordered, finite list of LSG's, no two of which have equal data (see below). The list is ordered so that if L_1 and L_2 are on the list and have data d_1 and d_2 respectively, then $d_1 < d_2$ if and only if L_1 is before L_2 in the list.

3b) A ULSG is an LSG.

4a) An intersect LSG (ILSG) is an arbitrarily ordered,* finite list of LSGs.

4b) An ILSG is an LSG.

5) The only LSGs are those defined by (1) - (4).

For various purposes, an LSG at any given time will be considered to be identifying a unique datum.

D3. The datum of an LSG is defined recursively as follows:

1) The datum of a PLSG is the datum of the list set element currently pointed at by the reader.

2) The datum of a DLSG, ULSG or ILSG is the datum of the first LSG of which it is composed.

---

*If the ILSG is ordered on the size of the sets to be generated by the component LSG's, smallest first, all operations on the ILSG will be significantly faster than otherwise.

$L_1 - L_2$

$L_1 = \{0,1,2,5,6,8,9\}$

$L_2 = \{0,1,2,3,7,9\}$

Step	DLSG	Generated Set
1	$(_D$ 1:0*)	{ }
2	$(_D$ 1:0,2:0)	{ }
3	$(_D$ 1:1,2:0)	{ }
4	$(_D$ 1:1,2:1)	{ }
5	$(_D$ 1:2,2:1)	{ }
6	$(_D$ 1:2,2:2)	{ }
7	$(_D$ 1:5,2:2)	{ }
8	$(_D$ 1:5,2:7)	{5}
9	$(_D$ 1:6,2:7)	{5,6}
10	$(_D$ 1:8,2:7)	{5,6}
11	$(_D$ 1:8,2:9)	{5,6,8}
12	$(_D$ 1:9,2:9)	{5,6,8}
13	The PLSG for L_1 finishes	

*A PLSG will be represented as a list set identifier followed by ":" followed by the datum of the PLSG.

Fig. 2: Example of a DLSG being used to generate a set which is the difference between two sets

$L_1 \cup L_2 \cup L_3 \cup L_4$

$L_1 = \{0,1,2,5,6,8,9\}$

$L_2 = \{0,2,3,4,5\}$

$L_3 = \{2,3,6,8,9\}$

$L_4 = \{0,1,2,3,7,9\}$

Step	ULSG	Generated Set
1	$(_U$ 1:0)	{ }
2	$(_U$ 1:0,2:2)	{ }
3	$(_U$ 1:0,2;2,3;3)	{ }
4	$(_U$ 1:0,4:1,2:2,3:3)	{0}
5	$(_U$ 4:1,2:2,3:3,1:5)	{0,1}
6	$(_U$ 2:2,3:3,1:5,4:7)	{0,1,2}
7	$(_U$ 3:3,2:4,1:5,4:7)	{0,1,2,3}
8	$(_U$ 2:4,1:5,3:6,4:7)	{0,1,2,3,4}
9	$(_U$ 1:5,3:6,4:7)	{0,1,2,3,4,5}
10	$(_U$ 3:6,4:7,1:8)	{0,1,2,3,4,5,6}
11	$(_U$ 4:7,1:8,3:9)	{0,1,2,3,4,5,6,7}
12	$(_U$ 1:8,3:9)	{0,1,2,3,4,5,6,7,8}
13	$(_U$ 3:9)	{0,1,2,3,4,5,6,7,8,9}
14	finishes	

Fig. 3: Example of a ULSG being used to generate a set which is the union of four sets

$$L_1 \cap L_2 \cap L_3 \cap L_4$$

$L_1 = \{0,1,2,5,6,8,9\}$

$L_2 = \{0,2,3,4,5\}$

$L_3 = \{2,3,6,8,9\}$

$L_4 = \{0,1,2,3,7,9\}$

Step	ILSG	Generated Set
1	($_I$ 1:0)	{ }
2	($_I$ 2:0.1:0)	{ }
3	($_I$ 3:2,2:0,1:0)	{ }
4	($_I$ 4:2,3:2,2:0 1:0)	{ }
5	($_I$ 4:2,3:2,2:2,1:0)	{ }
6	($_I$ 4:2,3:2,2:2,1:2)	{2}
7	($_I$ 4:3,3:2,2:2,1:2)	{2}
8	($_I$ 4:3,3:3,2:2,1:2)	{2}
9	($_I$ 4:3,3:3,2:3,1:2)	{2}
10	($_I$ 4:3,3:3,2:3,1:5)	{2}
11	($_I$ 4:7,3:3,2:3,1:5)	{2}
12	($_I$ 4:7,3:8,2:3,1:5)	{2}
13	($_I$ 4:9,3:8,2:3,1:5)	{2}
14	($_I$ 4:9,3:9,2:3,1:5)	{2}
15	The PLSG for L_2 finishes	

Fig. 4: Example of an ILSG being used to generate a set which is the intersection of four sets

$$(L_1 \cup L_2) - (L_3 \cap L_4)$$

L_1 = 2,3,6,8,9 L_3 = 0,1,2,5,6,8,9

L_2 = 0,1,2,3,7,9 L_4 = 0,2,3,4,5,8

Step	LSG	Generated Set
1.	($_D$($_U$ 1:2))	{ }
2.	($_D$($_U$ 2:0,1:2))	{ }
3.	($_D$($_U$ 2:0,1:2)($_I$ 3:0))	{ }
4.	($_D$($_U$ 2:0,1:2)($_I$ 4:0,3:0))	{ }
5.	($_D$($_U$ 2:1,1:2)($_I$ 4:0,3:0))	{ }
6.	($_D$($_U$ 2:1,1:2)($_I$ 4:2,3:0))	{1}
7.	($_D$($_U$ 1:2,2:3)($_I$ 4:2,3:0))	{1}
8.	($_D$($_U$ 1:2,2:3)($_I$ 4:2,3:2))	{1}
9.	($_D$($_U$ 2:3,1:6)($_I$ 4:2,3:2))	{1}
10.	($_D$($_U$ 2:3,1:6)($_I$ 4:3,3:2))	{1}
11.	($_D$($_U$ 2:3,1:6)($_I$ 4:3,3:5))	{1,3}
12.	($_D$($_U$ 1:6,2:7)($_I$ 4:3,3:5))	{1,3}
13.	($_D$($_U$ 1:6,2:7)($_I$ 4:8,3:5))	{1,3,6}
14.	($_D$($_U$ 2:7,1:8)($_I$ 4:8,3:5))	{1,3,6,7}
15.	($_D$($_U$ 1:8,2:9)($_I$ 4:8,3:5))	{1,3,6,7}
16.	($_D$($_U$ 1:8,2:9)($_I$ 4:8,3:8))	{1,3,6,7}
17.	($_D$ 2:9 ($_I$ 4:8,3:8))	{1,3,6,7}
18.	2:9	{1,3,6,7,9}
19.	finishes	{1,3,6,7,9}

Fig. 5: An example using all three LSGs

The datum of a DLSG or an ILSG may or may not be an element of the list set the LSG is generating. It will be, if the last operation performed by the LSG was initializing, incrementing, incrementing to or past a datum, or incrementing past a datum as these operations are described below. It may not be if the last operation was checking a datum against the LSG or some operation not defined here. The datum of a PLSG or a ULSG will always be an element of the list set being generated.

The operations described below, except initialization, may be performed repeatedly on an LSG in order to consider successive elements in a list set. The elements will be generated in the order used for ordering the list sets and once passed, an element will not again be the datum of the LSG. Thus, eventually, an LSG will have been moved past all the elements of the list set it generates. When this occurs, we say the LSG is finished. An LSG may finish during any of the operations described below, in which case the operation concludes, returning an appropriate flag. Instead of giving the finishing conditions in every algorithm below, we give them here once, since they are the same for all.

D4. 1) A PLSG finishes when an attempt is made to increment it when it already points at the last element of its list set.

2) 2) A DLSG finishes when its first LSG finishes.

3) A ULSG finishes when it is composed of one LSG and that LSG finishes

4) An ILSG finishes when any of its LSG's finishes.

In two cases an LSG may be discarded in favor of a component LSG: 1) If the second LSG of a DLSG finishes, the first LSG replaces the DLSG. 2) when a ULSG is composed of only one LSG, that LSG is used in place of the ULSG. These cases may also arise during any of the algorithms described below, but we will not mention them again.

The first algorithm to be described is initializing an LSG. When an LSG is initialized, its datum will be the first element of the list set the LSG generates. If that list set is null, the LSG will finish during the initialization process.

A1. Initializing an LSG

1) PLSG: Initialize the reader so that it points at the first element of its list.

2) DLSG (see Fig. 2 steps 1-8):

   a) Initialize the first LSG.

   b) Initialize the second LSG at or past the datum of the first (i.e. its datum will be equal to or larger than the datum of the first LSG).

   c) If the data of the two LSGs are equal, increment the DLSG.

3) ULSG (see Fig. 3 steps 1-4):

   Initialize each LSG in turn, placing them in the ULSG in the proper order (but not placing one that finishes). If an LSG is initialized with a datum equal to the datum of an LSG already in the ULSG, increment it until it has a datum not already

represented. Then place it in the ULSG in the proper order.

4) ILSG (see Fig. 4 steps 1-6):

a) Initialize one LSG and place it in the ILSG.

b) Initialize each successive LSG (in any order) at or past the datum of the previous LSG and place it as the first LSG of the ILSG.

c) When all LSGs have been initialized and inserted, if their data are not all equal, increment the ILSG to or past the datum of its first LSG.

Once an LSG is initialized, it can be repeatedly incremented, and after each step its datum will be the next greatest element of the list set it generates (see Figs. 2-4). If some operation was performed on an LSG so that its datum is not an element of the set it generates, and the LSG is then incremented, its datum after being incremented will be the smallest element of the set it generates which is larger than the datum of the LSG before it was incremented.

A2. Incrementing an LSG

1) PLSG: The reader is incremented so that it points at the next element in its list.

2) DLSG: a) Increment the first LSG.

b) Check the current datum of the first LSG against the second LSG. If the

. check succeeds, go back to step (a).

3) ULSG: a) Remove the first LSG from the ULSG.

b) Increment the LSG removed in step (a). If it finishes, the increment is done.

c) If the datum of the LSG incremented in step (b) is equal to the datum of any other LSG in the ULSG, go to step (b).

d) Return the LSG to the ULSG in its proper order according to its current datum.

4) ILSG: a) Increment the first LSG of the ILSG.

b) Let D be the datum of the first LSG and i be 1.

c) Let i = i + 1.

d) If there is no $i^{\underline{th}}$ LSG the increment is done.

e) Increment the $i^{\underline{th}}$ LSG to or past D.

f) If the datum of the $i^{\underline{th}}$ LSG equals D, go to step (c).

g) Let D be the datum of the $i^{\underline{th}}$ LSG and i be 0.

h) Go to step (c).

There are times when we are not interested in the next element to be generated by an LSG, but in the next element equal to or greater than a given one (e.g., A2 (4e)) or the next element strictly greater than a given one. Such an element could be found by repeatedly incrementing the LSG, but it would be more efficient to make full use of the

information of what datum we wish to equal or exceed and increment the LSG to or past (or just past) the datum in one operation.

A3. Incrementing an LSG (to or) past a datum, D

1) PLSG: Increment the reader (zero or more times) until it points to an element whose datum is (equal to or) greater than D.

2) DLSG: a) Increment the first LSG (to or) past D.

b) Check the datum of the first LSG against the second LSG. If the check is successful, increment the DLSG. If the check is not successful, the increment is done.

3) ULSG: a) Remove the first LSG from the list.

b) Increment the LSG removed in step (a) (to or) past D. If it is finished, go to step (e).

c) If the datum of the LSG incremented in step (b) is equal to the datum of any other LSG in the ULSG, increment it. If this finishes the LSG, go to step (e).

d) Return the LSG to its proper place in the ULSG according to its current datum.

e) If the datum of the LSG which is now first in the ULSG in not (equal to or) larger than D, go to step (a), otherwise the increment is done.

4) ILSG: This is exactly the same as incrementing an ILSG (A2 (4)), except that in step (a), the first LSG is incremented (to or) past D.

If it is desired to determine if a given element is a member of the set generated by an LSG, this can, in most cases, be done more quickly than by incrementing the LSG to or past the element and then looking at the datum of the LSG if it is acceptable that when the check is finished, the datum of the LSG might not be a member of the set it generates. To afterwards produce an unknown member of the set generated by the LSG, it would be necessary to perform one of the incrementing operations on it.

A4. Checking a datum, D, against an LSG

1) PLSG: Increment the reader to or past D. If the reader finishes or it ends up with a datum which is larger than D, the check is unsuccessful. If the PLSG ends up with a datum equal to D, the check is successful.

2) DLSG: a) Check D against the first LSG. If this check is unsuccessful, the check against the DLSG is unsuccessful.

b) If the check against the first LSG was successful, check D against the second LSG. If this check is successful, the check against the DLSG is unsuccessful and vice versa.

3) ULSG: a) If the datum of any LSG in the ULSG equals D, the check is successful.

b) Remove the first LSG from the ULSG.

c) Increment the LSG removed in step (b) to or past D. If it finishes, go to step (f).

d) If the datum of the LSG incremented in step (c) is equal to the datum of any other LSG in the ULSG, increment it. If this finishes the LSG, go to step (f).

e) Return the LSG to its proper place in the ULSG according to its current datum.

f) If the datum of the LSG incremented in step (c) was equal to D after it was incremented, the check is successful. Otherwise, if the datum of the LSG which is now first in the ULSG is larger than D, the check is unsuccessful. If neither of the above two cases hold, go to step (b).

4) ILSG: Check D against each LSG that makes up the ILSG in turn. As soon as one is found for which the check is unsuccessful, the check against the ILSG is unsuccessful. If all checks are successful, the check against the ILSG is successful.

It should be remembered that the list sets are ordered and the LSG's generate them in order. The only way to generate all the members of a set is by successive incrementing with no other operations interposed. An LSG cannot be "backed up" to an element it has already passed. If several elements are to be checked against an LSG, this must be done in the proper order to avoid the necessity of initializing several LSG's.

The sample problems of Fig. 5 and Appendix A show how LSG's are used to evaluate a set expression. Because of their generality, LSG's would be extremely useful as part of a language system allowing sets as a data type. Moreover, since any ordered, sequential file fits the definition of list set given above, LSG's may be used for traditional file handling and will lead to great efficiency when arbitrary numbers of files are to be handled simultaneously. For these purposes, the DLSG is used for purging records from a file, the ULSG is used for merging files* and all the LSG's may be used for information retrieval.

2.6.2 Extending LSGs to Handle LISs

We will show here how PLSGs, ULSGs and ILSGs may be extended to handle set expressions involving LISs** assuming that when taking the union of LISs all the ISs in the

---

*The ULSG merges in the manner described as "m-way merge with ranking sort" [8].

**The extension of DLSGs is not discussed because the only time a DLSG would be used in MENTAL would be in evaluating a speclist with the -spec option, and in that case every item that appears in the evaluated -spec should be removed from the main spec regardless of substitutions since the

LISs are ISs of the same general statement and that compatible intersections are wanted when intersecting LISs. We will also assume that list readers are used in such a way that if a reader is reading an LIS L and its datum is D, there is a field of the reader (which we shall call the SUBST field) that points to the substitution s for which D:s is an IS on L.

Each LSG has a unique substitution associated with its datum, which we shall call the datum-substitution of the LSG. They are:

PLSG -- The substitution pointed to by the SUBST field of the reader.

ULSG, ILSG -- The datum-substitution of the first component LSG.

We will now explain how the datum-substitutions are updated so that if the datum of an LSG is D and the datum-substitution is s, D:s is an IS on the LIS generated by the LSG.

PLSG -- When the reader is incremented, change the SUBST field appropriately.

ULSG -- When a component LSG is incremented so that its datum is D and its datum-substitution is s and D is the datum of another component LSG, L, with a datum-substitution s_1 change the datum-substitution of L to be the u-merge of

---

meaning of the ($spec_1$ - $spec_2$) speclist is "list every item that is described by $spec_1$ except those that can be described by $spec_2$".

$\underline{s}$ with $\underline{s}_1$. Operations should be performed on component LSGs so that every member of the set being generated by the component LSG gets its substitution handled by the above process.

ILSG -- When all component LSGs of the ILSG have the same datum, that datum is a member of the set being generated if and only if all the datum-substitutions of the component LSGs are compatible. If that is the case, change the datum-substitution of the first component LSG to be the i-merge of the datum-substitutions of all the component LSGs.

3. Using the Structure for Deductions

3.1 Representation of Deduction Rules

In Chapter 2 it was shown how the MENS structure is used for explicit storage and retrieval. In this chapter 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 we will use (according to the categorization of Section 1.4) will be deduction by executive routines and stored rules. Within this category we will use the method of storing general deduction rules and using 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. MENS also uses the deduction-by-structure method, which is a by-product of the explicit storage method described above.

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 [4, 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 2.2 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." [49, p. 118 quoted in 9, 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 excluded 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 [72] 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. In order for an item to be recognized as a variable, when it is pointed to from another item there is a specificity measure of 0 prefixed to its internal name. The specificity measure will be discussed more completely in Section 4.5.

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

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

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

That is if $\text{MUTIMP}(P_1,\ldots,P_n)$ 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, $1 = i = n$. For two arguments MUTIMP is equivalent to IFF.

relations 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 system relations emanating from it. They are:

(i) Q 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:

(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 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 son 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))
```

3.2 Interpreting Deduction Rules

3.2.1 Introduction

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

*This sentence taken from Green and Raphael [25].

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 substructures. Thus, several independent substructures might be deduced for 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 this section, 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.

3.2.2 Finding Deduction Rules

A deduction rule is needed when the number of items found to satisfy a findspec (see Section 2.5) is less than 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) $(?O,\#,L_1:(I_{11},\ldots,I_{1m_1}),\ldots,L_n:(I_{n1},\ldots,I_{nm_n}))$

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 findspec:

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

V is a list of all variable items in the data structure. For the sake of efficiency, we maintain an item which we shall

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 findspec (ii) is equivalent to:

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

Note that any item that satisfied (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 which has the independent statement flag (see Section 4.4). That item will be the head of a deduction rule capable of generating the consequences we are interested in. While following this path a trace list 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, $2 \leq i \leq 1$, 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 findspec 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.

3.2.3 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 given independent statement flags and left in the data structure or be erased (see Section 4.6 for a discussion of how this decision might be made). The Generate route 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 are described below. (Flowcharts are given in Appendix B.) They 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 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.

PREDGEN is a recursive routine that generates one instantiation of a substructure the internal name of whose head it is given. Each free variable in the substructure must have a current substitution. Since the substructure may contain quantifier clauses, the instantiation generated may itself be a deduction rule.

GENERATE -- Main Section is given the internal name of an item and transfers to the appropriate section to generate the consequences of the substructure the item heads. If the item is a constant, it is the consequence. If TRFL is on and TRACELIST is empty, we are at the statement to be generated and PREDGEN is called. If the item is a quantifier clause, ALLGEN or EXISTSGEN is transferred to, depending on where Q points. If the item is a connective clause, depending on where OP points,

the transfer will be made to NOTGEN, IMPGEN, IFFGEN, ANDGEN, ORGEN or MUTGEN. Otherwise, we are at a simple, non-quantified statement with free variables and PREDGEN is called to generate the instantiation. Other connectives or quantifiers may be added by adding an appropriate test here and an appropriate section similar to those described below.

NOTGEN changes NEGFL to the opposite of what it was. If TRFL is on, the top of TRACELIST must be the name of the item pointed to via the ARG pointer, so it is popped off. GENERATE is called recursively with the item pointed to via ARG as argument.

ALLGEN transfers to EXISTGEN (just after the point where EXISTGEN 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.

EXISTGEN transfers to ALLGEN (just after the point where ALLGEN tests NEGFL) if NEGFL is on. Otherwise, it creates a new item which is made the substitution for the variable of the quantifier. GENERATE is called with the restriction as

argument and with NEGFL and TRFL turned off, the substructures thus generated being stored as independent statements. Finally, NEGFL and TRFL are restored, TRACELIST is popped if TRFL is on, and GENERATE is called recursively with the scope as argument.

If at IMPGEN NEGFL is off, the statement is of the form P = > Q, and Q is generated if P is confirmed or the negation of P is generated if Q is refuted. If TRFL is on, only the one of the two above cases is tried that will result in generating the item at the top of TRACELIST. If NEGFL is on, we generate both P and Q unless TRFL is on, in which case only one of these is generated. If NEGFL is off but P is not confirmed and Q is not refuted, PREDGEN is called to generate the instantiation of P => Q since that will give the questioner as much information as possible.

If at IFFGEN NEGFL is off, if either argument can be confirmed (refuted) the other argument (the negation of the other argument) is generated. If NEGFL is on, confirmation of one argument leads to the generation of the negation of the other while refutation of one causes generation of the other. As in IFFGEN, if TRFL is on only the appropriate cases will be tried, and regardless of the states of TRFL and NEGFL if nothing is confirmed or refuted the entire statement is generated.

ANDGEN transfers to ORGEN (after its NEGFL test) if NEGFL is on. If TRFL is off, each argument pointed to via MARG is generated, but if TRFL is on, only the argument at the top of TRACELIST is generated.

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.

MUTGEN transfers to ORGEN as mentioned above if NEGFL is on. Otherwise, an attempt is made to confirm or refute each argument in turn (except for the one on top of TRACELIST if TRFL is on). As soon as one is confirmed, all are generated. If one is refuted, the negation of all the rest are generated. If TRFL is on, when an item is confirmed (refuted) only the item on top of TRACELIST is generated (has its negation generated).

3.2.4 Confirm and Refute

The Confirm and Refute routines are used by the Generate routine as explained above. It must be remembered that confirming or refuting a deduction rule does no more than show it to be true or false in the universe of the data structure as it exists at the time the confirmation or refutation is performed. The confirmation of a universally quantified expression

or the refutation of an existentially quantified expression may hold at one time but not hold after another individual fulfilling the proper restriction is introduced into the data structure. Furthermore, when individuals might be removed from the data (e.g. employees in a management information system), the refutation of a universally quantified expression or the confirmation of an existentially quantified expression may cease to hold after such a removal.

It is also possible for an expression to be neither confirmable nor refutable at some time. 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 $100^{\underline{th}}$ man.

The Confirm and Refute routines are given the internal name of the head of a substructure all of whose free variables have substitutions and return TRUE if the substructure is confirmed or refuted, respectively, and FALSE otherwise. They are outlined in parallel below.

Assume I is the argument.

1. If I has the independent statement flag, I is confirmed.
2. If I is negated and has the independent statement flag, I is refuted.
3. If I has a Q pointer to ALL
   a. Find all substitutions for the variable satisfying the restriction.
   b. For each substitution for the variable, try to

confirm or refute the scope.

c. If the scope is confirmed for each substitution, I is confirmed.

d. If the scope is refuted for one substitution, I is refuted.

4. If I has a Q pointer to EXISTS

a. Find all the substitutions for the variable satisfying the restriction.

b. For each substitution for the variable, try to confirm or refute the scope.

c. If the scope is confirmed for one substitution, I is confirmed.

d. If the scope is refuted for each substitution, I is refuted.

5. If I has an OP pointer to NOT

a. If the argument is refuted, I is confirmed.

b. If the argument is confirmed, I is refuted.

6. If I has an OP pointer to IMPLIES, calling the first argument P and the second Q

a. If P is confirmed and Q is confirmed, I is confirmed.

b. If P is confirmed and Q is refuted, I is refuted.

c. If P is refuted, I is confirmed.

7. If I has an OP pointer to IFF, calling the arguments P and Q

a. If P and Q are both confirmed, I is confirmed.

b. If P and Q are both refuted, I is confirmed.

c. If one of P and Q is confirmed and the other is refuted, I is refuted.

8. If I has an OP pointer to AND
   a. If all arguments are confirmed, I is confirmed.
   b. If one argument is refuted, I is refuted.
9. If I has an OP pointer to OR
   a. If one argument is confirmed, I is confirmed.
   b. If all arguments are refuted, I is refuted.
10. If I has an OP pointer to MUTIMP
   a. If all arguments are confirmed, I is confirmed.
   b. If all arguments are refuted, I is confirmed.
   c. If at least one argument is confirmed and at least one argument is refuted, I is refuted.
11. If none of the above cases holds
   a. Use the substructure to search memory for a substructure isomorphic to that one, but containing the proper substitution items for the variables. (Use SDFIND, described below.)
   b. If such a substructure is found with an independent statement flag, I is confirmed.
   c. If such a substructure is found negated with an independent statement flag, I is refuted.

3.2.5 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 (Section 2.5). 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 the reason substitutions and instantiation specifications were developed in Section 2.5. Thus, whenever an instantiation is discovered for a substructure within a general substructure we keep track of the instantiation specification. That is, with the head item S, we associate the substitution s the variables of whose components are the non-constant items in the general substructure and whose terms are the appropriate substituting items of the instantiation. Using the compatible intersection operation we ensure that in the instantiation substructure not more than one item substitutes for any one item in the general substructure.

The procedure for finding instantiating substructures for a general substructure is outlined below. S is the head of a general substructure and LO is a list of items that have been examined in the general substructure before the present recursive call to the procedure and so need not be examined again. Once an LIS is discovered for an item, it is recorded so

that if that item is met again in the general substructure the LIS need not be rediscovered. If a variable already has a substitution or a list of possible substitutions, they are used. Thus, this procedure can be used for checking a list of possible substitutions against a restriction as well as for finding all substitutions for a variable satisfying a restriction.

SDFIND

1. Let LO1 = LO U (S).

2. If there are no more system relations from S to be considered, go to 14.

3. Let R be the next system relation from S.

4. If R is a reverse scope, restriction or argument pointer, go to 2.

5. If there are no more items pointed at via R, go to 2.

6. Let I be the next item pointed at via R.

7. If I is a constant, put the list *R(I) on list L and go to 5.

8. If I is a variable that has a substitution, IS, put *R(IS):(I/IS) on the list L, then go to 5.

9. If I is a variable that has a list of possible substitutions $(PS_1,\ldots,PS_n)$, put $(*R(PS_1):(I/PS) U \ldots U *R(PS)_n : (I/PS_n))$ on the list L, then go to 5.

10. If I is a variable and has neither a substitution nor a list of possible substitutions, put I on top of the list L1, then put R on top of L1, then go to 5.

11. If I is on the list LO1, go to 5.

12. If I has an LIS recorded for it, put *R(LIS), where LIS is the recorded l.i.s., on the list L, then go to 5.

13. Call SDFIND recursively with I and LOl as arguments, getting an l.i.s. $(S_1:\underline{s}_1,\ldots,S_n:\underline{s}_n)$. Put

$$(S_1:\underline{s}_1,;\ldots,S_n:\underline{s}_n)\cap(S_1:(I/S_1,\ldots,S_n\ (I/S_n))$$

on the list L, also record it as the l.i.s. for I, then go to 5.

14. L will be a list of LISs, calling the resultant LIS LI.

15. LI is an LIS of the form $(S_1:\underline{s}_1,\ldots,S_n:\underline{s}_n)$. LIS is a list of the form $(R_1,V_1,\ldots,R_m,V_m)$. Form the compatible intersection of LI with

$$(S_1:(V_1/R_1(S_1),\ldots,V_m/R_m(S_1)),\ldots,S_n:(V_1/R_1(S_n),\ldots,V_m/R_m(S_n))).$$

Return the resultant l.i.s. as the value of SDFIND.

With the aid of SDFIND, finding all substitutions for a variable, V, that satisfy a restriction and checking a list of possible substitutions against a restriction become the same procedure, except that for the latter the list of possible substitutions is nonempty. SDFINDC (outlined below) is called with two arguments, the first being the variable for which substitutions are to be found or checked, the second being the head of the general structure. SDFINDC takes account of any quantifiers and connectives in the structure and makes use of SDFIND for the non-complex expressions.

SDFINDC(V,I)

1. If I has a Q pointer to ALL

   a. If NEGFL is on, go to 2b.

b. For each item in SDFINDC(VB(I),R(I)), set the item as the substitution of VB(I) and put SDFINDC(V,S(I)) on the list L.

c. Return the intersection of all lists on L.

2. If I has a Q pointer to EXISTS

a. If NEGFL is on, go to 1b.

b. Set SDFINDC(VB(I),R(I)) as the list of substitutions for VB(I).

c. Return SDFINDC(VB(I),R(I)).

3. If I has an OP pointer to NOT

a. Change NEGFL to the opposite of the way it is set.

b. Return SDFINDC(V,ARG(I)).

4. If I has an OP pointer to IMPLIES

a. Let L = SDFINDC(V,ARG2(I)).

b. Change NEGFL.

c. Return the union of L with SDFINDC(V,ARG1(I)).

5. If I has an OP pointer to IFF

a. Let TF = NEGFL.

b. Let L1 = SDFINDC(V,ARG1(I)).

c. Let L1 = L1 $\cap$ SDFINDC(V,ARG2(I)).

d. Let NEGFL = $\neg$TF.

e. Let L2 = SDFINDC(V,ARG1(I)).

f. Set NEGFL on.

g. Let L2 = L2 $\cap$ SDFINDC(V,ARG2(I)).

h. Return L1 $\cup$ L2.

6. If I has an OP pointer to AND

a. Let TF = NEGFL.

b. For each item, S, on MARG(I)

i) Set NEGFL to TF.

ii) Put SDFINDC(V,S) as a sublist on L.

c. If TF is on, return the union of the sublists on L.

d. If TF is off, return the intersection of the sublists on L.

7. If I has an OP pointer to OR

a. Let TF = NEGFL.

b. For each item, S, on MARG(I)

i) Set NEGFL to TF.

ii) Put SDFINDC(V,S) as a sublist on L.

c. If TF is on, return the intersection of the sublists on L.

d. If TF is off, return the union of the sublists on L.

8. If I has an OP pointer to MUTIMP

a. Let TF = NEGFL and turn NEGFL off.

b. For each item, S, on MARG(I)

i) Put SDFINDC(V,S) as a sublist of L1.

ii) Set NEGFL.

iii) Put SDFINDC(V,S) as a sublist of L2.

c. If TF is on

i) Let L1 be the union of the sublists on L1.

ii) Let L2 be the union of the sublists on L2.

iii) Return L1 $\cap$ L2.

d. If TF is off

i) Let L1 be the intersection of the sublists

on L1.

ii) Let L2 be the intersection of the sublists on L2.

iii) Return L1 U L2.

9. If none of the above cases hold

a. Let L = SDFIND(I,())

b. If NEGFL is on, remove from L all ISs, S<s, for which S is not negated.

c. If NEGFL is off, remove from L all i.s.'s, S:s for which S. is negated.

d. Return a list of all items in the term of some substitution component whose variable is V and which is an IS in L.

4. Contributions Toward Solution of Some Open Problems for Question-Answering Research

4.1 Introduction

This chapter consists of discussions of several topics relevant to question-answering systems and the general problem of computer understanding of natural language. These topics comprise major problems in the area. The discussions present either the way the solution of the problem is envisioned in MENS and MENTAL or an approach that may be taken in looking for a solution using MENTAL as a research vehicle.

4.7 Relevance

A major problem in question-answering systems and in information retrieval systems in general is the one of relevance -- how to identify in a large corpus of data the information relevant to a given question or topic. MENS has been designed with the relevance problem in mind. Thus, it is a net structure whose nodes are joined into substructures when they are involved in the same fact or event. In this way, items relevant to each other will tend to have short paths between them in the net and items that are irrelevant to each other will tend to be connected only by long paths if they are connected at all. We can actually identify three kinds of relevance which we may call relevance with respect to a concept, with respect to a discussion and with respect to a question.

When we are interested in relevance with respect to a concept, we are interested in those other concepts the understanding of which aids the understanding of the given concept. This is the sort of relevance we mean when we ask for topics related to a given topic, or ask for a discussion of a concept with discussions of related concepts, or ask for two related concepts to be compared and contrasted (as Quillian's Semantic Memory [47] does). This kind of relevance is provided in MENS by the path between items.

Relevance with respect to a discussion includes the questions of whether a concept is relevant to a previous discussion, which of several concepts (e.g., possible senses of an ambiguous word) is most relevant to an on-going discussion or whether a sentence is relevant to the discussion at hand. In MENS this kind of relevance may be determined by the paths between two substructures or between an item and a substructure.

Relevance with respect to a question is a factor in determining what information is useful for answering a question the answer to which is not explicitly stored. Information relevant in this sense is found in MENS by using the deduction rules in the manner described in Section 3.2. Both the rules themselves and especially the restrictions insure the relevance of the substructures that are found.

All three kinds of relevance depend in MENS on the information supplied to the system by the human informant, so his judgments are reflected in the system's. The determination of all three kinds of relevance is further aided by

considering only that subnet of MENS which is in the same universe of discourse as the concepts in question. Universes of discourse and their use in MENS are discussed in Section 4.4.

4.3 Disambiguation, Anaphora and the Discourse File

It may be the case that the number of items which match a speclist exceeds the num given in the findprefix as the maximum number of items to be retrieved. This would occur if the speclist were not a complete enough description of what was wanted, or, perhaps, if several items were, in fact, identically connected into the net. The latter case does not really involve an ambiguity since the items are synonymous, but the former is a case of semantic ambiguity and must be resolved.

There are four ways MENTAL could resolve ambiguities, analagous to four ways a human would handle the same problem. Let us say someone mentions "John's brother" to MENTAL and to a human. Both MENTAL and the human listener know that John has two brothers, Bill and Henry, and that Bill is in high school but Henry is in college. The human might say, "Do you mean Bill or Henry?" and the speaker might respond with "Henry." Similarly, MENTAL might output a message saying that the request was ambiguous with Bill and Henry both fitting the speclist and request the user to enter a list whose length was within the maximum num and whose elements were taken from the list of items actually found. This list would then be used as the value of the speclist.

Instead of replying, "Henry," the speaker might have said, "I mean the one in college." To MENTAL, the user might reply with another speclist giving additional information. The value of this speclist would then be intersected with the ambiguous value of the original speclist and the result, if within the maximum num, would be the final value. If the intersection of the two lists were still too large, MENTAL would ask for additional information. Actually, the first method of resolving the ambiguity is a special case of this one, since a list of names is a speclist.

The above two methods involve simply asking the speaker (user) to resolve the ambiguity, but a human listener might be able to resolve the ambiguity himself if he has been listening to the entire conversation. Perhaps Henry himself had been under discussion or had otherwise just been mentioned, in which case the listener might assume that Henry was meant. Perhaps the discussion had been about college students, in which case also the listener might assume that Henry was meant. MENTAL could also use these methods if it had a discourse file.

A discourse file is a subset of the items in the net or a list of internal names of a subset of the items in the net. This subset consists of the items that are relevant to the discussion at hand. Relevant items include those that have been mentioned in the current discussion since the last change of topic. They may also include items of known interest to the speaker and items related to the topic under discussion even though they have not been

explicitly mentioned. In human terms, we may expect a speaker to discuss certain topics and use certain concepts because we know who he is, whom he knows and what his interests are. For example, we would interpret the term "set" in one sense if uttered by a mathematician, in another sense if uttered by a psychologist and in still another sense if uttered by a tennis pro. Yet, if the mathematician were discussing his tennis game, we would interpret his use of "set" in the tennis sense. In the MENTAL system the discourse file could be started with the known interests of the user (see Section 4.4 for how MENS would be divided into subnets by user interest), and then updated and revised as the conversation develops. This revision could be done by adding newly mentioned concepts to the file and removing those that have not been mentioned for a while.

The two methods of disambiguation using the discourse file would be checking the actual contents of the file (in our example, noting that Henry was being discussed) and finding items closely related to items in the file (see the discussion of relevance with respect to a discussion in Section 4.2).

These same methods can also be used for determining anaphoric references, such as finding what man is referred to in the sentence, "Then the man turned and walked away." Actually, if we consider anaphoric references, we find them, in general, to be ambiguous descriptions of things. In the above example some man is referred to, but it is not clear from the one sentence which man is meant. Similarly,

pronouns provide some information as to what they refer to (e.g., gender) but not enough when out of context. The assumption that an anaphoric reference succeeds in referring to an individual is equivalent to saying that the context provides enough clues for the reference to be understood, and this indicates that MENTAL may handle anaphoric references as ambiguities to be disambiguated from the discourse file.

4.4 The Independent Statement Flag and Categorization Pointers

There are several system relations that should be used in MENS, but which are seldom mentioned elsewhere in this paper because their pervasiveness would only clutter the descriptions. They are the independent statement flag, and the two net catagorization pointers -- the source pointer and the universe of discourse pointer.

The independent statement flag is termed a flag because it is a single pointer which, when present, always points to the same item. It indicates that the item from which it emanates heads a substructure which may be output to the user or otherwise used without any qualification. That is, the independent statement flag indicates that the item it is attached to represents some information that has been alleged to be true. Two other cases are possible -- the information may have been alleged to be false (e.g., "It is not true that the sky is green.") or no allegation may have been made as to its truth value (e.g., the statement

"the sky is green" in the sentence "Henry says the sky is green"). The independent statement flag is needed so that only information the user considers true is stated to him or used for generating consequences for him. If the human user enters information using the input syntax of Section 2.5, he must enter the independent statement flag explicitly. If a parsing program is used as an interface between the human and MENTAL, it should be responsible for adding the independent statement flag to the user's sentences and to subordinate clauses where appropriate (e.g., to the statement "Prince John was tall" in the sentence "Prince John, who was tall, became king.").

The categorization pointers are used to divide the net into subnets and thus reduce search time, avoid seeming, but not actual, contradictions, and aid disambiguation. They should be used when several people are using MENTAL with the same data file but independently of each other and/or when some person is using MENTAL with the same data file for several different fields of knowledge. The categorization pointers divide the net such that the smallest subnets are specific to one user and one field of his interest. Search time is reduced in the cases where a spec that is part of another spec may match several substructures in the entire net and fewer substructures in the subnet determined by including the proper categorization pointers in the spec. Two subnets may contain contradictory information either because two users have conflicting beliefs or because two fields have different logical systems. These apparent

conflicts will not matter if the simultaneous use of the two subnets is avoided by using categorization pointers. Disambiguation would be easier because many ambiguous terms or descriptions would have a unique referent in a given subnet. The categorization pointers can be used to initialize the discourse file (see Section 4.3).

Although, as categorization pointers, both the source pointer and the universe of discourse pointer provide the capabilities mentioned, there is a reason for their distinction.

The source pointer should provide the major divisions of the net. The source of a substructure should be the one who is responsible for the substructure, i.e., responsible for its accuracy and its consistency with other substructures he is responsible for. For example, if Henry enters the statement that John says something, Henry should be the source for "John says ..." if Henry is taking the responsibility for it. In that case John may neither know nor care that the information is being entered. If, however, Henry is acting as John's agent or as a neutral reporter of John's statements, John may be made the source of what he said. This, however, places the responsibility for the statement's being in the file on John. When someone uses the file, he would want the information used to answer his questions to be limited to that for which he is responsible or that for which someone he trusts and believes is responsible. In this way a partial ordering could be established among users to control information flow in an organization.

Thus, source pointers can help to keep information private, i.e., to limit access to it. It would be possible to use private information to deduce public information without compromising the private information. For example, let G be a completely reliable "source" whose information can be used but not output, and say John has private opinions about which people in his organization are overweight, but is willing to give out for use the fact that all the overweight people were at some meeting. The following deduction rule could be entered into the system: "According to G, all the people who, according to John, are overweight were, according to anyone, at the meeting." This will allow anyone to get a list of people at the meeting without divulging John's private opinions.

Another use for the source pointer is to keep track of the assumption and deduction rules upon which a deduced substructure is based. The reason for doing this is that there is nothing to stop a user from entering inconsistent information into the data structure. He may do this unknowingly, and may discover contradictions at a later time. It would then be useful to discover the source of the contradiction and remove it. Once this is done, all substructures deductively based on the removed information should then be found and rechecked for validity.

The universe of discourse pointer provides the minor divisions in the net. It derives from the discussion that has been going on for some time among philosophers [10,64-68;12;42,83-86;50,177 ff,246;51,2-9,103,165;68,176]

about the proper analysis of "things" that do not exist in the real world, such as the present king of France, square circles, and Pegasus. As is pointed out by the ordinary language philosophers [12;42;68], there is a difference between the sentences that might be uttered about square circles and those about Pegasus in that the former are all equally meaningless whereas the latter may be true or false. The difference is not only that between logical contradictions and nonrealized possibilities since there is a universe of discourse, viz. Greek mythology, in which there is a "body of knowledge" about Pegasus. Cartwright [12] considers statements such as those about Pegasus to be statements about unreality and states, "unreality is just that: it is not another reality" [12, p 66]. Yet there is a need to distinguish several different unrealities. For example, the characters in "Peanuts" are as unreal as Pegasus, yet, to Charlie Brown, Pegasus is (presumably) as mythological as he is to us. Thus, it would be useful to distinguish many universes of discourse (for example, different ones for different works of fiction), each of which contained individuals and facts about them that would not exist in the other universes of discourse. The universe of discourse pointer is used to keep these various bodies of universe-relative facts as well as various fields of knowledge or activity in the real world separate yet allow them to be entered and properly used.

The proper universe of discourse for any substructure would most accurately be determined by the human user, but it would be possible to have a program try to determine it

using a measure of relevance with respect to a discussion in order to find that discussion to which the substructure seems most relevant. It might also be possible to devise deduction rules that could deduce the proper universe of discourse for a substructure.

4.5 Specificity

When those pieces of information that are relevant to a given topic are identified, there often remains the problem of deciding which relevant piece of information is most specific to the topic, for that will be where one wants to direct his attention first. In MENTAL, this problem emerges mainly when several deduction rules are found that might be capable of generating some desired consequence. For example if we wanted to know if it were true that "Elizabeth rules England" we might find any of the eight statements of Fig. 6 as possible consequences of deduction rules. Each of these eight is clearly relevant to our problem, but that of Fig. 6a is just as clearly most specific. Those of Fig. 6b are somewhat less specific. Those of Fig. 6c are even less specific, and that of Fig. 6d is least specific. It seems reasonable that we should try the deduction rules in the same order, viz. those of Fig. 6a then b then c then d. This is because the more specific rules are more likely to be successful, and even if they are not, they are likely to require less work to check.

The question, then, is how, after locating the general substructures we would like to generate, can we order them according to specificity. We notice that the version of specificity we have been discussing (there are others) involves the percent of the total number of items in a substructure that are variables. We also recall that all the lists in MENS are ordered on internal names of items, largest first. If we can compute a specificity measure (SM) for each item as it is created and

a.

ELIZABETH RULES ENGLAND

---

b.

ELIZABETH RULES X

ELIZABETH X ENGLAND

X RULES ENGLAND

---

c.

ELIZABETH X Y

X RULES Y

X Y ENGLAND

---

d. X Y Z

Fig. 6: Relevant statements differing in number of variables

prefix that measure to the internal name of the item for use in pointing to that item from others, then all lists, including the lists of general statements produced by the search for deduction rules, will have their major sort on specificity. Thus we will get orderings by specificity for very little additional cost. The drawback to such a scheme is that once an SM is computed it should not be changed; or should be changed very seldomly, because a change would require changing the entry in all multiple pointer lists that point to the item in question and reordering these lists. This, however, is not a severe drawback, since the SM mentioned above will not change if it is handled in the following way:

1. The SM for a constant item is established as a high number, m_c, when the item is first introduced into the data structure.

2. The SM for a variable item is established as 0 when the item is first introduced into the data structure.

3. Every other item is introduced by a buildspec which specifies every item to which the new item points and which should partake in the calculation of the new item's SM. The new item's SM is established as the average of the SM's of the items it points to according to the buildspec specifying its creation at the time of its creation.

The SM of an item will only be recorded with its internal name in items that point to it. This is no problem since all constant items will have the same SM as will all variable items, and any other item may be mentioned by a user only by giving a

findspec for it, so that its name will be found via pointers and, thus, its proper SM will be found and used.

There is another version of specificity that we can recognize and incorporate with the SM discussed above. This specificity involves levels of structure. Among several items whose SM's as calculated above are equal, it would seem that one heading a substructure involving more levels is more specific than one heading a substructure with fewer levels. For example, say we wanted to generate a substructure of the form (P∨Q)=>(A&B), where P,Q,A,B are some items, and we find the general statements in Fig. 7 as possible consequences of some deduction rules. Again, these are all relevant to the problem (as they are guaranteed to be by the method of finding them), but that of Fig. 7a is most specific, that of Fig. 7d is least specific and those of Fig. 7b and c are intermediate with b being more specific than c. This idea of specificity may be combined with the previous one by changing step 3 above so that after the average is taken a small constant, m_s, is added. Thus the SM for the structures of Fig. 7 would be:

a. $\frac{5m_c + 6m_s}{9} + m_s$

b. $\frac{4m_c + 3m_s}{9} + m_s$

c. $\frac{m_c}{3} + m_s$

d. 0

The actual structure for Fig. 7a is shown in Fig. 8 with the specificity for each item shown.

a.

$(X \vee Y) \Rightarrow (W \& Z)$

---

b.

$(X \vee Y) \Rightarrow Z$

$X \Rightarrow (Y \& Z)$

---

c.

$X \Rightarrow Y$

---

d.

X

Fig. 7: Relevant statements differing in amount of structure

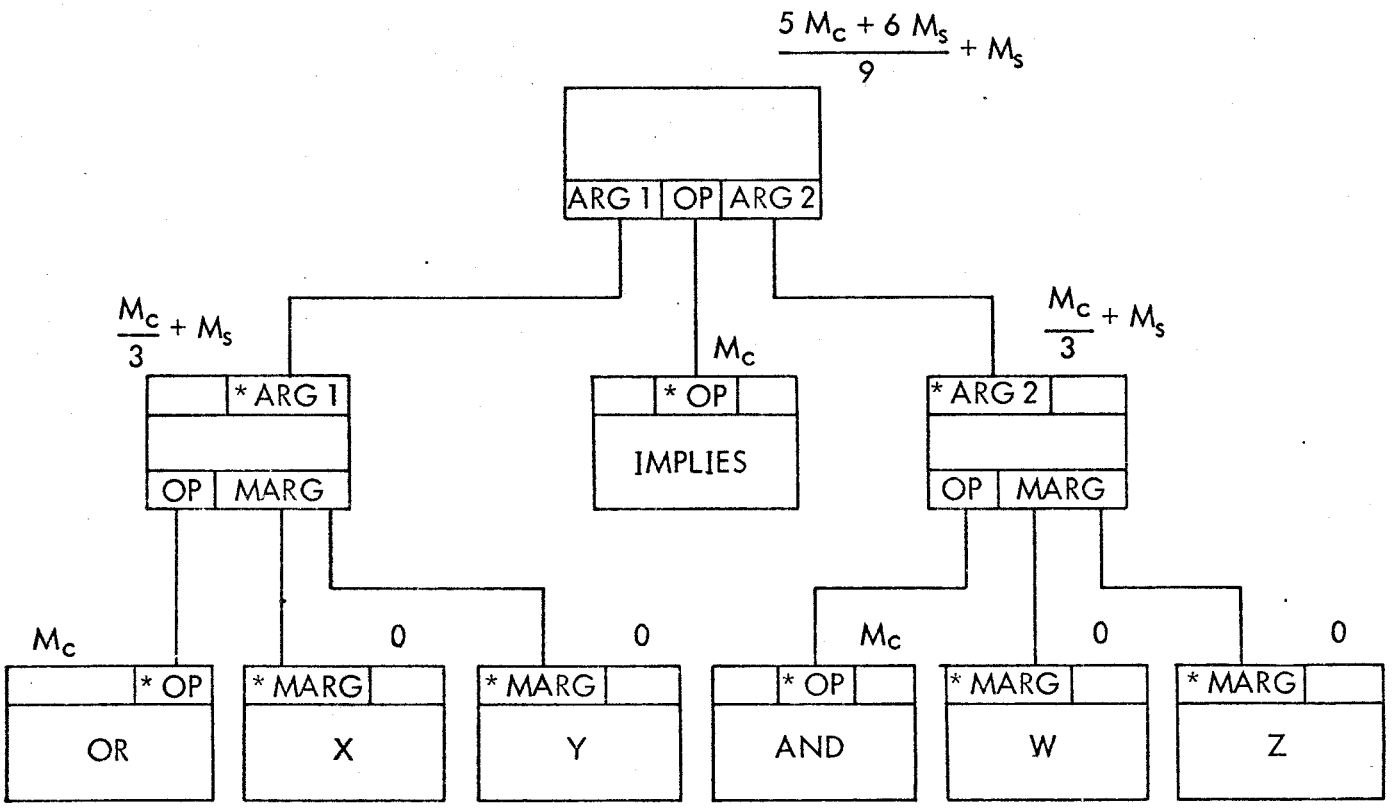


Fig. 8 — A structure with specificity measures

It may be that there are additional versions of specificity that might be incorporated into the SM. These and other aspects of specificity present themselves as a fruitful area for future research.*

---

*For example, it would be possible for the user to declare a weight for each system relation when he declares it and for this weight to be used to form a weighted average in step 3 above instead of the strict average given.

4.6 Recognizing Significant Results

Once a consequence has been generated through the use of a deduction rule, a decision must be made to keep it as an independent statement or to erase it from the data structure. This is basically a question of a space-time tradeoff -- the space needed to store the consequence vs. the time needed to regenerate it if it is needed again. A prior question arises, however, since the data structure is presumed to be incomplete and growing (or even changing). This question is, "Will the consequence still be true at any later time?"

As mentioned in Sec. 3.2.4, a consequence generated by confirming a universally quantified expression or by refuting an existentially quantified one may be false after another individual fulfilling the appropriate restriction is introduced into the data structure. Keeping such a consequence would be a case of improper generalization, which people often do by not examining all the members of a class about which they then draw conclusions. Consequences of this sort must either not be stored or be stored with some indication of the state of the data structure when they were generated. In either case, unless all consequences are handled the same way, these special ones must be recognized, which may be a difficult task.* The problem of consequences becoming false because

*Although the task may not appear difficult, the experience of J. L. Kuhns should be kept in mind. In /38/, he discusses the problem of characterizing those questions that are "unreasonable" to ask a computerized question-answerer. The main problem being questions whose answers change when the dictionary of known individuals changes although the store of facts remains constant. He proposes a class of definite

of individuals being removed from the data structure is, of course, similar to the above problem caused by adding individuals.

Leaving this problem, let us consider those consequences which we can trust to be true in the future. How can we decide whether or not to keep them in the data structure? As stated above, this is a question of a space-time tradeoff. If a lot of work were required to deduce the consequence, we would be more likely to want to keep it than if only a small amount of work were required. Also if the consequence seems to be particularly useful or significant we would want to keep it more than if it were useless and unimportant. Saving a consequence would provide the system with increased knowledge on hand, making it, as it were, more of an expert on the field of knowledge the consequence deals with and shortening the deductive processes needed to generate further consequences based on that one.

The amount of work needed to generate a consequence should include both the actual time taken and the number of deduction rules and specific statements employed. One or both of these must be kept track of anyway so that the user can be checked with when a deduction takes too long (remember that it is possible for a statement to be true but not deducible in the system). Also, the statements upon which the consequence is

---

formulas as the class of "reasonable" questions. R. A. DiPaola later showed /17/ that no algorithm can be written to decide for any formula whether or not it is definite.

based must be recorded so that if one of them is later denied or removed from the structure, the consequence can be called into question.

A more interesting problem is measuring the significance of a consequence. It seems reasonable that the significance of a fact is directly related to its usefulness in deducing other facts. Thus, deduction rules would be more significant than specific facts. In fact, we could say that one aspect of significance is the inverse of the specificity, which is calculated for each item as described in Sec. 4.5. Another aspect of significance is the number of deduction rules for which the fact is a substitution instance of a restriction or of a clause of the scope. In each of these cases the fact can be used for generating other facts. For example, if a fact is the negation of a substitution instance of one of a set of disjuncts, it has some significance because it allows the elimination of a possible case. Whenever deduction rules are being considered or counted for calculating significance, recursive deduction rules should be counted more heavily than non-recursive ones. A recursive deduction rule is one that is capable of generating a consequence which can be used in that same deduction rule to generate other consequences, or which can be used in some other deduction rules to generate consequences which can be used in other deduction rules, etc. eventually generating a consequence which can be used in the original deduction rule. Actually all deduction rules can be weighted with a weight directly proportional to their own

significance (previously calculated). Individual items can also be measured for significance according to the number of deduction rules in which they appear.

Thus the value, V, of keeping a consequence in the data structure is some formula of the form aW + bS, where a and b are constants, W is the amount of work required to generate it, and S is its significance.

The measure V has several uses other than deciding what consequences to keep in a data store. In a theorem prover, a line of the proof which has a large V may be a useful lemma. If V is high enough, the line may be an interesting theorem on its own. In programs to generate interesting theorems cf. Slagle /67/, V is a good measure for which theorems are interesting. In a computer aided instruction program that generates its own questions, a fact with a high V is a good one to change to question form and ask. Clearly, further work on developing measures of significance would be useful.

5. Uses To Represent and Explore Semantic Theories

The MENS structure is intended to represent the meanings of the terms that are represented by its constant items. It embodies a theory of semantics in the same sense as this is done in Quillian's "Semantic Memory" /47/ and "TLC" /48/ and in the Protosynthex programs /58;64;65/ (cf. /23/). In being a conceptual structure, MENS fits the class of theories promulgated by the cognitive psychologists. For example, Ausubel states, "According to the cognitive structure view, new meanings are acquired when potentially meaningful symbols, concepts, or propositions are related to, and incorporated within, a particular individual's cognitive structure on a nonarbitrary, substantive basis" /3,p74/. Thus, in MENS, as in the systems mentioned above, the meaning of a word is the totality of its relationships with the entities represented by the items in the net, including its capabilities to partake in the generation of new substructures as embodied in the deduction rules that refer to it.

It is not, however, claimed that MENS represents one semantic theory in its entirety. It is, rather, an approach and an environment in which various theories may be tested. Theories testable on MENTAL may vary as to what the items are used to represent, what the system relations are, the design of substructures to represent natural language text, how extensively deduction rules are used and the design of specific deduction rules. The method for using MENTAL as an experimental system is to attempt to store a large corpus of general

information in MENS using some theory on how information from natural language sentences may be represented. The corpus should be derived from an actual natural language text and deduction rules should be devised and entered so that proper consequences can be drawn from the information in the text. The theory may immediately be tested on the criteria of ease of translating natural language sentences into its representation, and value of the representation for organizing the information from several sentences and for deducing sentences that should be deducible from given sentences. The rest of this chapter is devoted to a variety of suggestions and questions in current linguistic theory and how they may be realized or investigated in the MENTAL system.

The examples in this paper have utilized the case grammar approach as discussed by Fillmore /21/. Although case grammars seem to be an approach to the problem of syntax, they may be viewed as an approach to semantics by considering the cases to be semantic relations between the terms of a sentence and the fact or event represented by the sentence. It is thus satisfying and not surprising that all the following sentences (and more) may be generated from the structure whose buildspec* is

(.VERB:OPEN,AGENT:JOHN,OBJECT:DOOR,INSTRUMENT:KEY)

i) John opens the door with the key.

ii) John uses the key to open the door.

iii) John opens the door.

---

*Remember that relspecs may be used to define a system relation for each case (see Sec. 2.5).

iv) The key opens the door.

v) The door is opened by John.

vi) The door is opened with the key.

This view that syntactic deep structures are really semantic representations is currently held by various linguists, e.g., McCawley following Lakoff and Ross /39/ argues that "the syntactic and semantic components of the earlier theory will have to be replaced by a single system of rules which convert semantic representation through various intermediate stages into surface syntactic representation" /44, p167/ and Fillmore states, "I have the feeling that real progress can be made in understanding ... the semantic and syntactic properties of the major parts of speech, by abandoning a conception of syntax that restricts itself to categories and sequences in favor of a conception of syntax-semantics that is based on a theory of the essential ways in which aspects of linguistically codable experiences are relatable to each other and to the experience as a whole" /22, p393/.

An open question in case grammar theory is what cases are required /21, p24-25/. Examining this question from the point of view of MENS, we see that the main problem is identifying the various relationships that might exist between a term and a fact or event. These relationships are candidates for being cases. A set of them must be chosen* that best satisfy the criteria mentioned above for a theory being tested on MENTAL.

---

*It may be that instead of finding one universally adequate (correct?) set of cases, various sets will be found, each adequate (good enough) for a particular context.

There are actually three ways of representing such a relationship in MENS -- as a system relation (case) of the sentence describing the fact or event, as a system relation in a sentence about the sentence, or as an item relation which is the verb of a sentence about the sentence. For example the locative relationship between an event and the place in which the event occurred might be handled in any of the three ways. The sentence "John kissed Mary in Chicago." might be entered into MENS with either of the three buildspecs (assuming system relations as appropriate):

i) (.VERB:KISS,AGENT:JOHN,OBJECT:MARY,LOCATIVE:CHICAGO)

ii) (.OBJECT:(.VERB:KISS,AGENT:JOHN,OBJECT:MARY),LOCATIVE:CHICAGO)

iii) (.VERB:LOCATION,AGENT:(.VERB:KISS,AGENT:JOHN,OBJECT:MARY),OBJECT:CHICAGO)

The resultant structures are shown in Figs. 9a, 9b and 9c respectively. These representations differ in the deductions they allow to be drawn. For example, if we wanted to store the rule that whenever two events take place at the same place they are "collocative", this might be done as follows if the representations of (ii) or (iii) are used respectively:

iv) (.Q:ALL,VB:%X,S:(.Q:ALL,VB:%Y,R:(.OBJECT:'X,LOCATIVE:'Y),
S:(.Q:ALL,VB:%Z,R:(.OBJECT:'Z,LOCATIVE:'Y),
S:(.VERB:COLLOCATIVE,MARG:'X,MARG:'Z))))

v) (.Q:ALL,VB:%X,S:(.Q:ALL,VB:%Y,R:(.VERB:LOCATION,AGENT:'X,

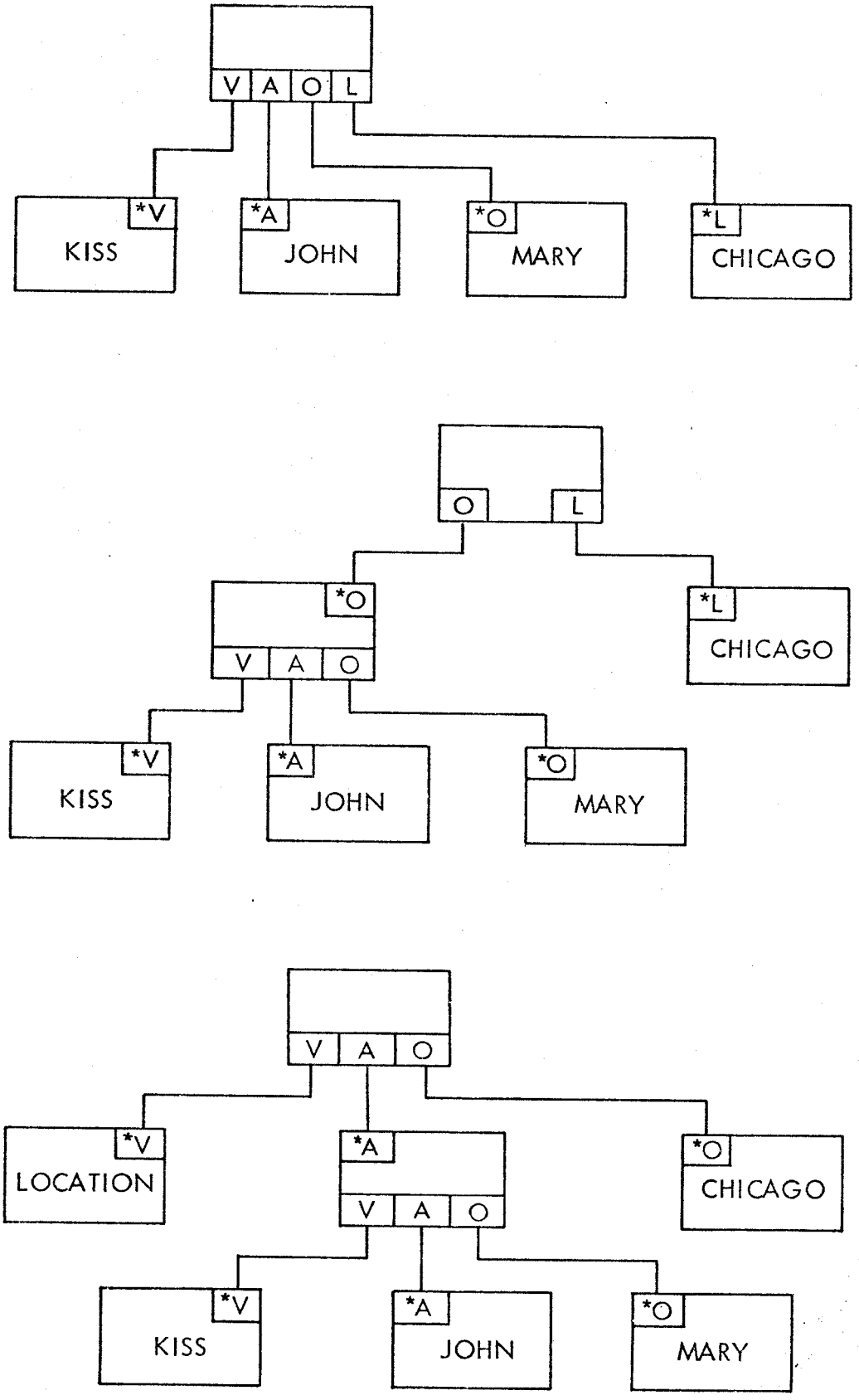


Fig. 9—Different structures for showing location

OBJECT:'Y),S:(.Q:ALL,VB:%Z,R:(.VERB:LOCATION,AGENT:
'Z,
OBJECT:'Y),S:(.VERB:COLLOCATIVE,MARG:'X,MARG:'Z))))

An equivalent rule could not be expressed if the representation of (i) were used since it would require a statement of the form $\forall x \forall y$(y is a term in the locative case of statement $x \supset \ldots$) and this cannot be expressed in MENTAL because it violates the rule that a system relation cannot be a term of a sentence. If we wanted to store the information that location is transitive so that the deduction rule for transitive relations could be used with location, we would have to use the representation of (iii) since it is the only one in which location is an item. The temporal relationship between an event and the time interval in which it occurs as well as other relationships could be examined in a similar manner.

Although Fillmore sees negation as appearing in the modality constituent of the sentence /21, p23/, it seems more productive to make it an operator on the sentence using a containing sentence, since if it were within the sentence, the theorem $\forall P(P \vee \neg P)$ could not be stored as a deduction rule. Perhaps all modalities should be handled this way.

Bach [4] discusses the view that nouns enter sentences through relative clauses and presents as an example the sentence, "The professors signed a petition" /p97/. His claim is that this derives from the underlying sentence, "Those who are professors signed that which is a petition" and that this is shown by the three interpretations of the negation of the sentence, viz.:

i) The professors didn't sign a petition.

i.e. It is not true that the professors signed a petition.

ii) The professors didn't sign a petition.

i.e. Those that signed a petition aren't professors.

iii) The professors didn't sign a petition.

i.e. The professors signed something that isn't a petition.

Bach's analysis may be represented in MENS as in Fig. 10. The three negative sentences may then be derived from this structure by negating the item labelled 1, 2, or 3 respectively. Fillmore suggested as a fourth possibility /4, p98fn/

iv) The professors didn't sign a petition.

i.e. The professors did something, which was not signing, to a petition.

To include this possibility, the original sentence would be structured as in Fig. 11, and the item labelled 4 would be negated to represent sentence (iv).

Another example Bach gives /4, p115/ is, "The man is working." which, he says, should be analyzed as

i) The one who is a man is working.

thus being similar to

ii) The one who is working is a man.

The differences in surface structure "result from transformational rules and are not to be attributed to the deep structures." In Fig. 12 is a MENS structure to represent this sentence such that if item 1 is taken as the head, sentence (i)

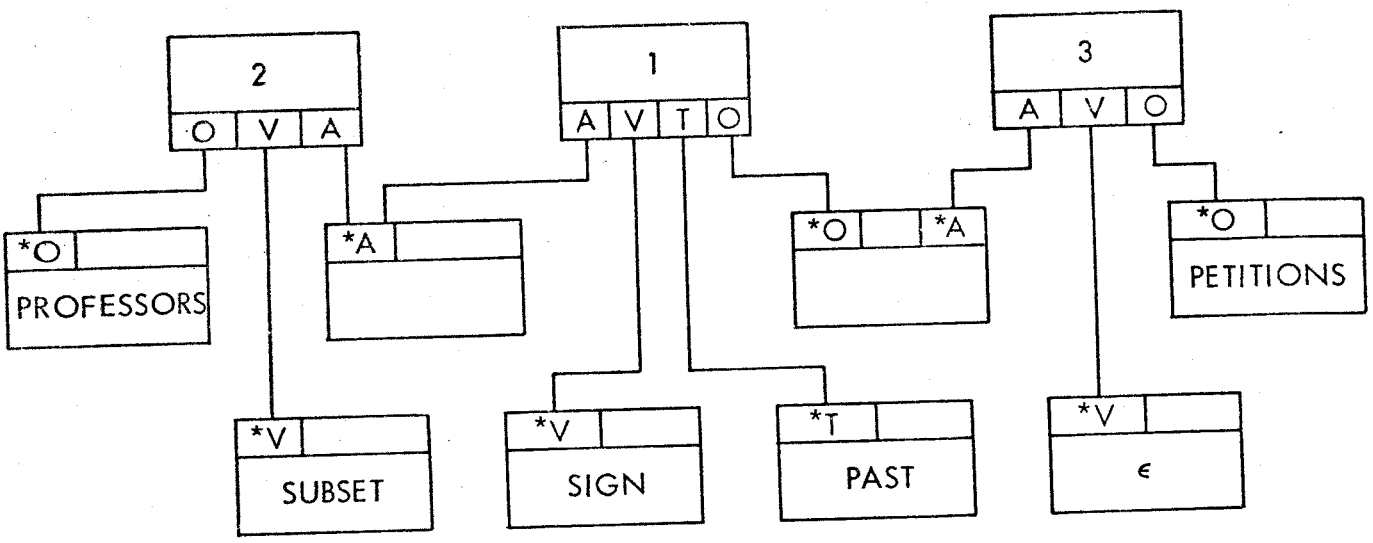


Fig.10 — A structure with three places for negation

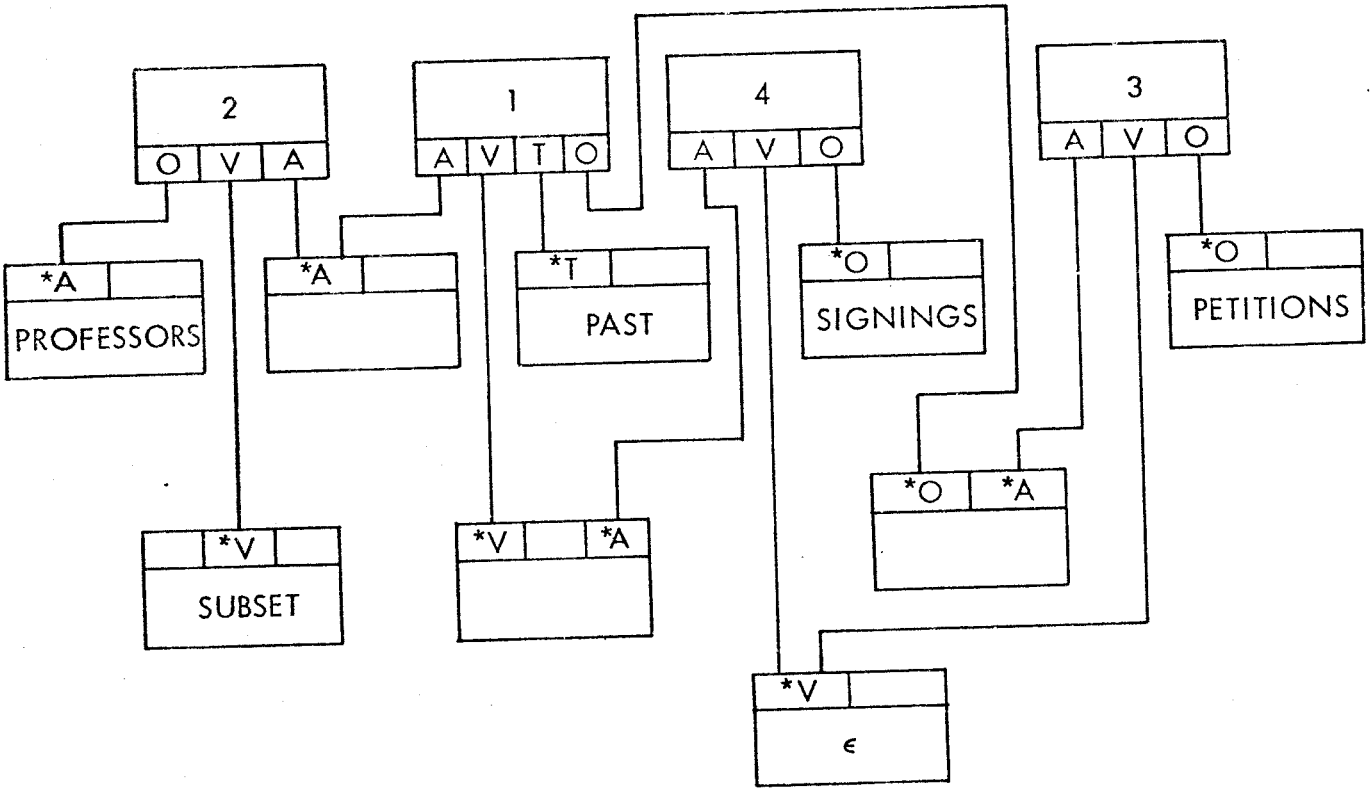


Fig.11 — A structure with four places for negation

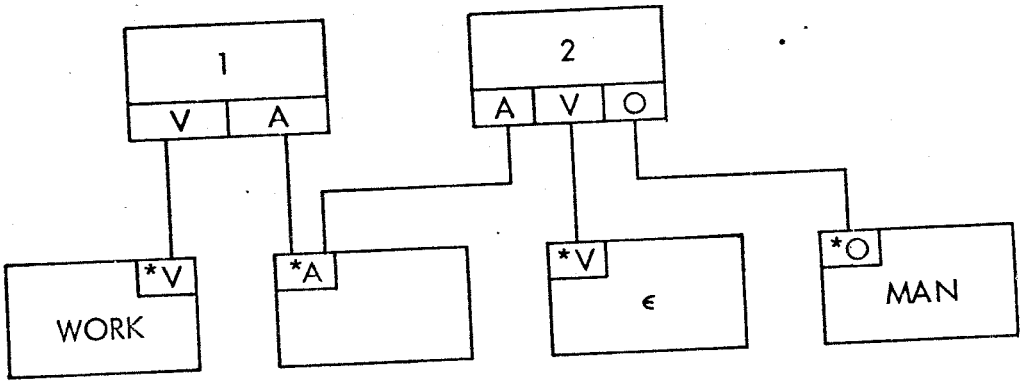


Fig.12 — A structure allowing choice of main and subordinate clause

results, while if item 2 is taken as the head, sentence (ii) results.

Lyons' meaning relations /43/ of incompatibility, antonymy (gradable and non-gradable), hyponomy, and conversity may be handled within a MENS semantic structure along with a theory for the representation of facts and events (such as a case grammar) by providing a deduction rule for each meaning relation. For example incompatibility could be represented by a deduction rule whose buildspec is:

```
(.Q:ALL,VB:%P,R:(.OP:INCOMPATIBLE,MARG:'P),S:(.Q:ALL,
    VB:%Q,R:(.OP:INCOMPATIBLE,MARG:'P,MARG:'Q),S:(.
    Q:ALL,VB:%X,R:(.VERB:'P,AGENT:'X),S:(.OP:NOT,ARG:
    (.VERB:'Q,AGENT:'X)))))
```

To demonstrate the possible use of MENS with a semantic theory totally unlike case grammars, we will show how Schank's conceptual dependency diagrams /57/ may be represented in MENS. Referring especially to Schank's list of conceptual rules /57, p189-90/, we might decide to represent his theory as follows.

Let items represent named concepts and instances of named concepts. An instance of a named concept has the CNCP system relation to the named concept it is an instance of. All arrows connect instances of named concepts or have either or both ends at a horizontal or vertical double barred double headed arrow. Therefore we must have a substructure for the horizontal or vertical double barred double headed arrow relational statement. Let HDD be the item representing the horizontal

double barred double headed arrow and VDD be the item representing the vertical double barred double headed arrow. Then we will have the system relation ARG1 point to the left or upper item, ARG2 to the right or lower item and OP to HDD or VDD. We will also let this substructure have the M system relation to either NEG, COND or INTER for negative, conditional and interrogative respectively and the TNS system relation to either FUT, PST, PR, or FAC for future, past, present or timeless fact respectively. All other arrows will be represented by system relations thusly:

VSS for vertical single barred single headed

VSD for vertical single barred double headed

HDS for horizontal double barred single headed

HSS for horizontal single barred single headed

VDS for vertical double barred single headed

VTS for vertical triple barred single headed

VBS for vertical dotted bar single headed

The relspecs defining all these system relations are:

```
$ CNCP S *CNCP M
$ ARG1 S *ARG1 S
$ ARG2 S *ARG2 S
$ OP S *OP M
$ M S *M M
$ TNS S *TNS M
$ VSS S *VSS S
$ VSD S *VSD S
$ HDS S *HDS S
```

$ HSS S *HSS S

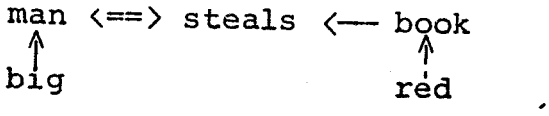
$ VDS S *VDS S

$ VTS S *VTS S

$ VBS S *VBS S

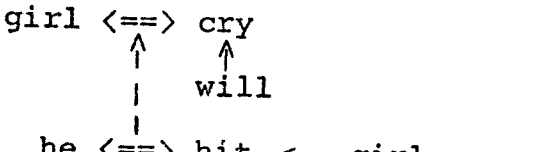
We will now give examples of several sentences from /57/, showing Schank's conceptual dependency diagrams and our buildspecs.

1) The big man steals the red book. /57, p56-7/



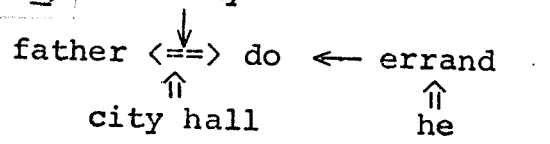
(.OP:HDD,ARG1:(.CNCP:MAN,VSS:(.CNCP:BIG)),ARG2:
(.CNCP:STEALS,HSS:(.CNCP:BOOK,VSS:(.CNCP:RED))))

2) If he hits the girl she will cry. /57, p67/



(.OP:HDD,ARG1:(.CNCP:GIRL),ARG2:(.CNCP:CRY,VSS:
(.CNCP:WILL)), VBS:(.OP:HDD,ARG1:(.CNCP:HE),
ARG2:(.CNCP:HIT,HSS:(.CNCP:GIRL))))

3) My father did his errand at city hall today. /57, p73/



(.OP:HDD,TNS:PST,ARG1:(.CNCP:FATHER),ARG2:(.CNCP:DO,
HSS:(.CNCP:ERRAND,VDS:(.CNCP:HE))),VSS:(.CNCP:TODAY),
VDS:(.CNCP:CITY HALL))

We have shown how various semantic theories may be represented in MENS. MENTAL may be used for extensive testing

of such theories on the criteria of ease of use for representing natural language sentences and value for organizing information to allow proper deductions.

6. 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 2.2. 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 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 structure, 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 computer understanding may productively be investigated. Also MENS and MENTAL may be used as an experimental vehicle for further research in semantic structures.

Appendix A

We will step through setting up and using an LSG (see Section 2.6) to evaluate the expression:

$((L_1 \cup L_2) \cap (L_3 \cup L_4 \cup L_5) \cap (L_6 \cup (L_7 \cap L_8) \cup L_9)) - (L_{10} \cup (L_{11} - L_{12}))$

where:

$L_1 = \{2\}$

$L_2 = \{3,4\}$

$L_3 = \{0,1,2,5,6,8,9\}$

$L_4 = \{0,2,3,4,5\}$

$L_5 = \{2,3,6,8,9\}$

$L_6 = \{4\}$

$L_7 = \{0,1,2,3,7,9\}$

$L_8 = \{4\}$

$L_9 = \{1,7,8\}$

$L_{10} = \{3,7\}$

$L_{11} = \{0,2,4,5,8\}$

$L_{12} = \{2,3,5,6\}$

1.	Initialize DLSG
1.1	Initialize ILSG
1.1.1	Initialize ULSG for $(L_1 \cup L_2)$
1.1.1.1	Initialize PLSG for L_1 , getting 1:2
1.1.1.2	Put in ULSG, getting $(_U$ 1:2)
1.1.1.3	Initialize PLSG for L_2, getting 2:3
1.1.1.4	Put in ULSG, getting $(_U$ 1:2,2:3)
1.1.2	Put ULSG in ILSG, getting $(_I(_U$ 1:2,2:3))

1.1.3 Initialize next ULSG at or past 2

1.1.3.1 Initialize PLSG for L_3 at or past 2, getting 3:2

1.1.3.2 Put in ULSG, getting $({}_U\ 3{:}2\ )$

1.1.3.3 Initialize PLSG for L_4 at or past 2, getting 4:2

1.1.3.4 Put in ULSG -- but equal to datum already there

1.1.3.4.1 Increment 4:2, getting 4:3

1.1.3.4.2 Now put in ULSG, getting $({}_U\ 3{:}2,4{:}3\ )$

1.1.3.5 Initialize PLSG for L_5 at or past 2, getting 5:2

1.1.3.6 Put in ULSG -- but equal to datum already there

1.1.3.6.1 Increment 5:2, getting 5:3

1.1.3.6.2 Still equal to datum already in ULSG

1.1.3.6.3 Increment 5:3, getting 5:6

1.1.3.6.4 Put in ULSG, getting $({}_U\ 3{:}2,4{:}3,5{:}6\ )$

1.1.4 Put ULSG in ILSG, getting $({}_I({}_U 3{:}2,4{:}3,5{:}6)({}_U 1{:}2,2{:}3))$

1.1.5 Initialize next ULSG at or past 2

1.1.5.1 Initialize PLSG for L_6 at or past 2, getting 6:4

1.1.5.2 Put in ULSG, getting $({}_U 6{:}4)$

1.1.5.3 Initialize ILSG for $(L_7 \cap L_8)$ at or past 2

1.1.5.3.1 Initialize PLSG for L_7 at or past 2, getting 7:2

1.1.5.3.2 Put in ILSG, getting $({}_I 7{:}2)$

1.1.5.3.3 Initialize PLSG for L_8 at or past 2, getting 8:4

1.1.5.3.4 Put in ILSG, getting $({}_I 8{:}4,7{:}2)$

1.1.5.3.5 Increment the ILSG to or past 4, so datum is a member of the set it generates

1.1.5.3.5.1 Increment 7:2 to or past 4, getting 7:7 and $({}_I 8{:}4,7{:}7)$ for the ILSG

1.1.5.3.5.2 Increment 8:4 to or past 7, finishing the PLSG for L_8, which finishes the ILSG -- the intersection was null

1.1.5.4 The ILSG was finished before being initialized so initialize a PLSG for L_9 at or past 2, getting 9:7

1.1.5.5 Put in ULSG, getting $({}_U 6{:}4,9{:}7)$

1.1.6 Put the ULSG in the ILSG, getting $({}_I({}_U 6{:}4,9{:}7)({}_U 3{:}2,4{:}3,5{:}6)({}_U 1{:}2,2{:}3))$

1.1.7 The ILSG is now built, but its datum is not in the set it generates, so increment it to or past 4, the datum of its first sub-LSG

1.1.7.1 Increment the 2nd ULSG to or past 4

1.1.7.1.1 Increment 3:2 to or past 4, getting 3:5

1.1.7.1.2 Replace it in the ULSG, getting $({}_U 4{:}3,3{:}5,5{:}6)$

1.1.7.1.3 Increment 4:3 to or past 4, getting 4:4

1.1.7.1.4 Replace it in the ULSG, getting $({}_U 4{:}4,3{:}5,5{:}6)$

1.1.7.2 The ILSG is now $({}_I({}_U 6{:}4,9{:}7)({}_U 4{:}4,3{:}5,5{:}6)({}_U 1{:}2,2{:}3))$ so increment the 3rd ULSG to or past 4

1.1.7.2.1 Increment 1:2 to or past 4, finishing the PLSG for L_1, so the ULSG is now $({}_U 2{:}3)$

1.1.7.2.2 Increment 2:3 to or past 4, getting 2:4

1.1.7.2.3 Note the ULSG is just $({}_U 2{:}4)$, so change it to the PLSG 2:4

1.1.7.3 The ILSG is now $({}_I({}_U 6{:}4,9{:}7)({}_U 4{:}4,3{:}5,5{:}6)\ 2{:}4)$ so it is initialized and its datum is 4

1.2 Initialize the 2nd LSG of the DLSG to or past 4

1.2.1 Initialize PLSG for L_{10} at or past 4, getting 10:7

1.2.2 Put it in the ULSG, getting $({}_{U}10{:}7)$

1.2.3 Initialize DLSG for $(L_{11} - L_{12})$ at or past 4

1.2.3.1 Initialize PLSG for L_{11} at or past 4, getting 11:4

1.2.3.2 Initialize PLSG for L_{12} at or past 4, getting 12:5

1.2.3.3 The DLSG is now initialized as $({}_{D}11{:}4,12{:}5)$

1.2.4 Put the DLSG in the ULSG, getting

$({}_{U}({}_{D}11{:}4,12{:}5)\ 10{:}7)$

1.3 The DLSG is now

$({}_{D}({}_{I}({}_{U}6{:}4,9{:}7)({}_{U}4{:}4,3{:}5,5{:}6)\ 2{:}4)({}_{U}({}_{D}11{:}4,12{:}5)\ 10{:}7))$

so 4 is the datum of both LSGs in the DLGS and therefore not an element of the set generated by the DLSG, so the DLSG must be incremented past 4

1.3.1 Increment the ISLG past 4

1.3.1.1 Increment $({}_{U}6{:}4,9{:}7)$ past 4

1.3.1.1.1 Increment 6:4 past 4, finishing the PLSG for L_6

1.3.1.1.2 The ULSG is now $({}_{U}9{:}7)$ and the ILSG is

$({}_{I}({}_{U}9{:}7)({}_{U}4{:}4,3{:}5,5{:}6)\ 2{:}4)$

1.3.1.2 Increment $({}_{U}4{:}4,3{:}5,5{:}6)$ to or past 7

1.3.1.2.1 Increment 4:4 to or past 7, finishing the PLSG for L_4

1.3.1.2.2 Increment 3:5 to or past 7, getting 3:8

1.3.1.2.3 Replace it in the ULSG, getting $({}_{U}5{:}6,3{:}8)$

1.3.1.2.4 Increment 5:6 to or past 7, getting 5:8

1.3.1.2.5 Replace it in the ULSG -- but datum already there

1.3.1.2.5.1 Increment 5:8, getting 5:9

1.3.2.5.2 Now replace it in the ULSG, getting $({}_{U}3{:}8,5{:}9)$

1.3.1.3 The ILSG is now $({}_{I}({}_{U}9{:}7)({}_{U}3{:}8,5{:}9)\ 2{:}4)$ so increment $({}_{U}9{:}7)$ to or past 8

1.3.1.3.1 Increment 9:7 to or past 8, getting 9:8

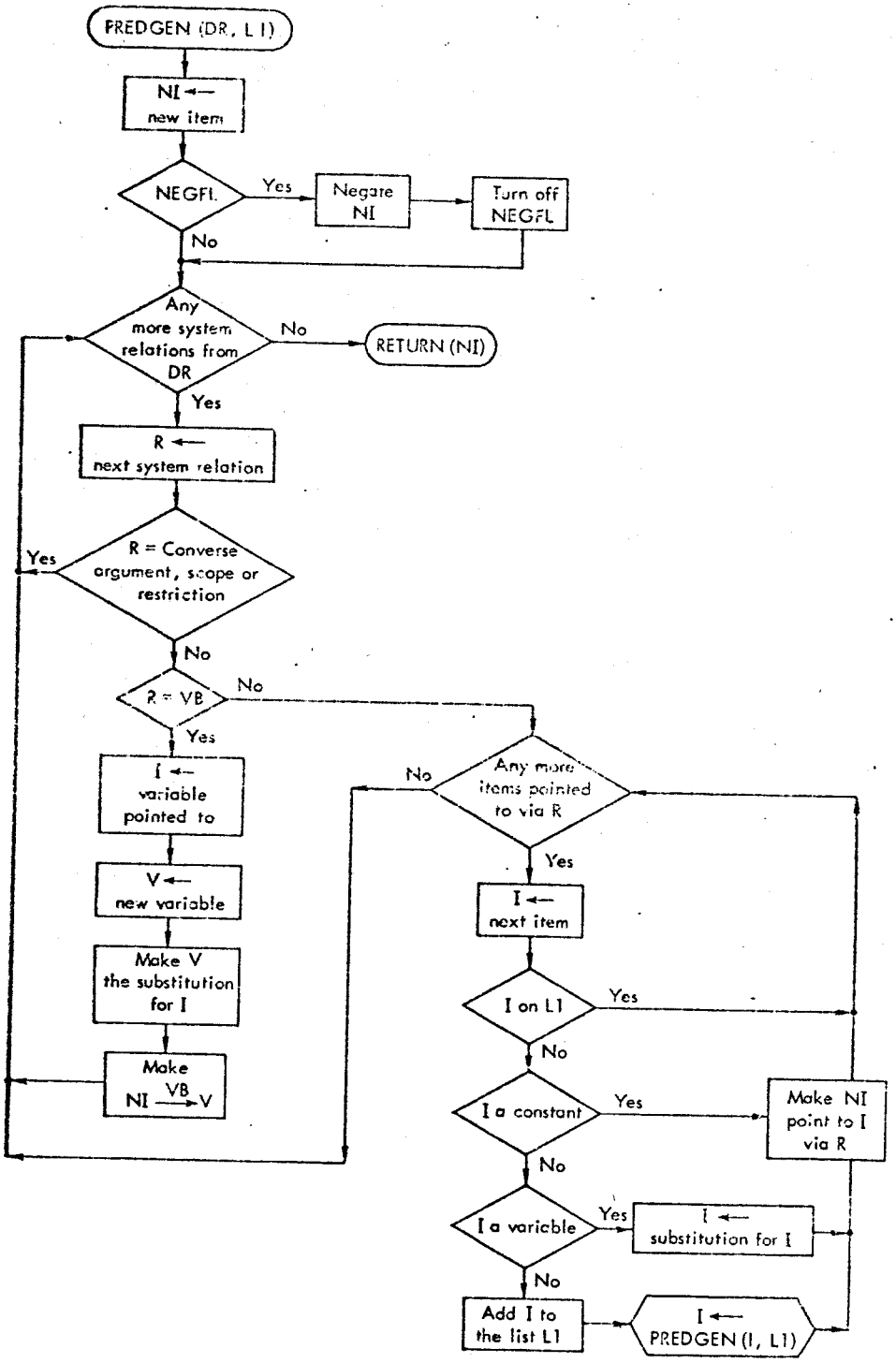
1.3.1.3.2 The ULSG is now just the PLSG 9:8

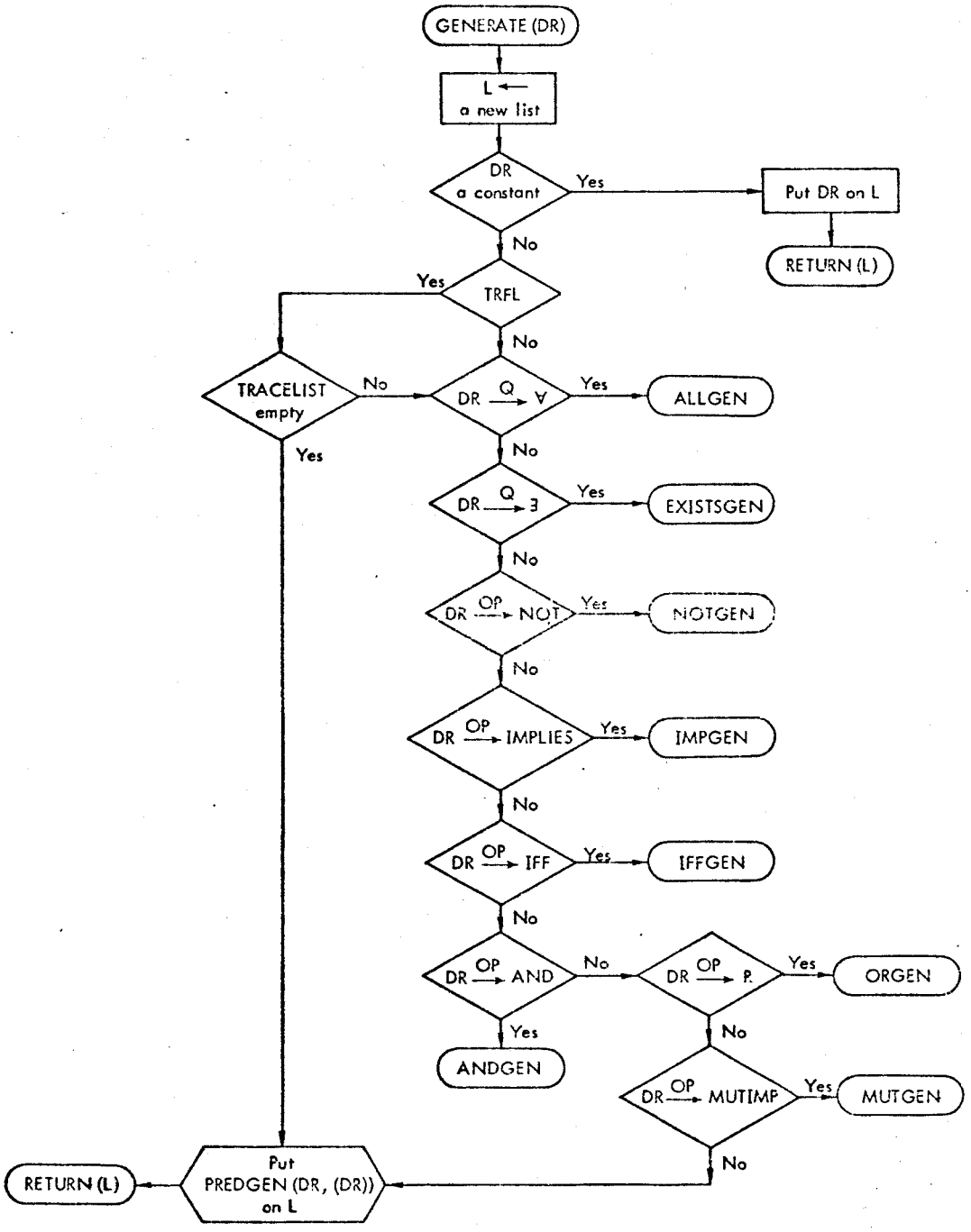
1.3.1.4 The ILSG is now $({}_{I}9{:}8\ ({}_{U}3{:}8,5{:}9)\ 2{:}4)$ so increment 2:4 to or past 8, finishing the PLSG for L_2, which finishes the ILSG, which is the first LSG of the DLSG so that is now finished

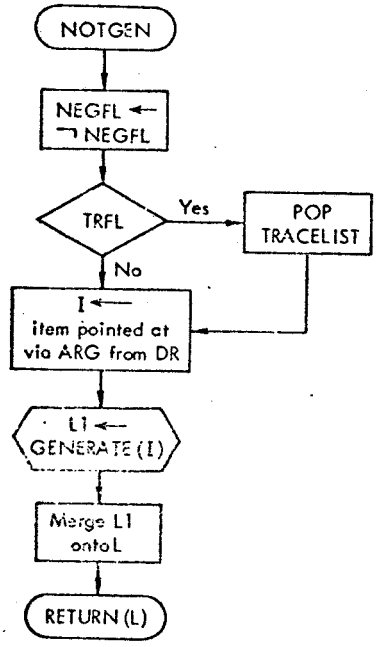
2 The DLSG was finished while being initialized. The value of the expression is the null set.

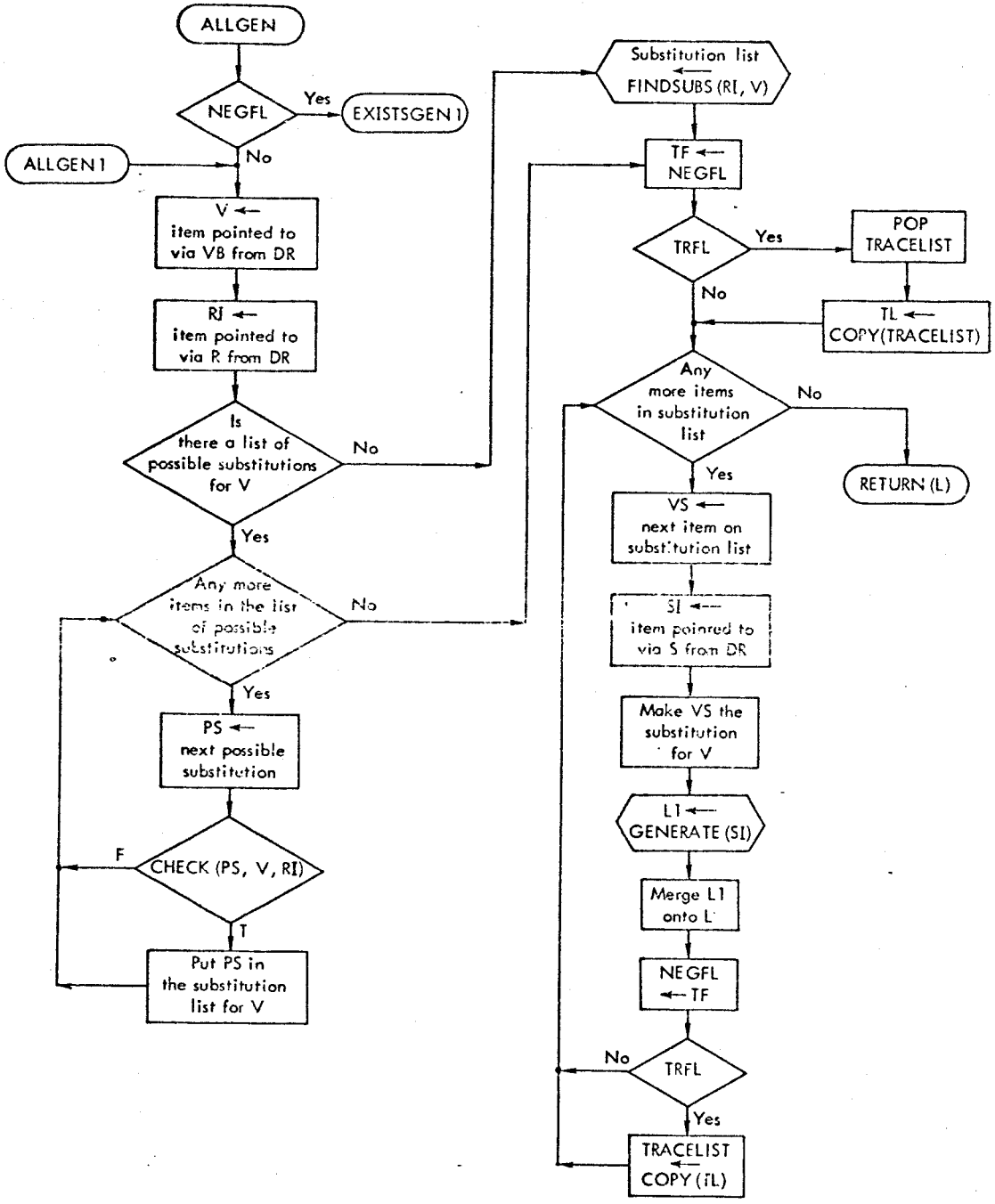
Appendix B

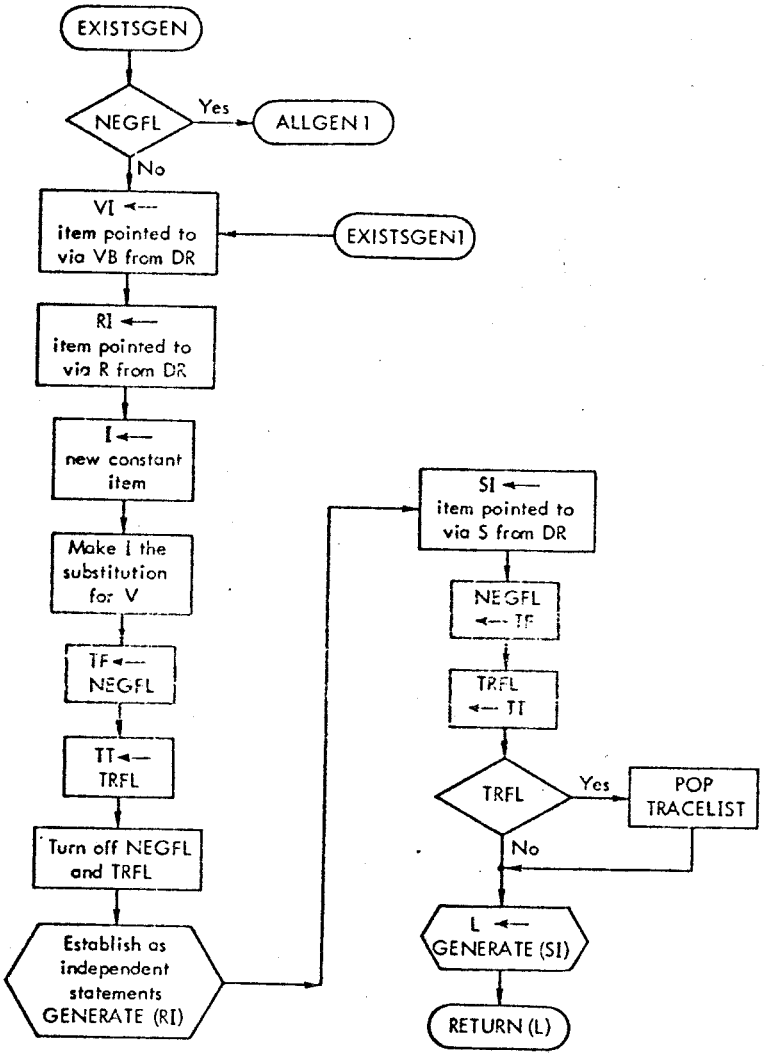
In this Appendix are the flowcharts of the GENERATE routines discussed in Section 3.2.3.

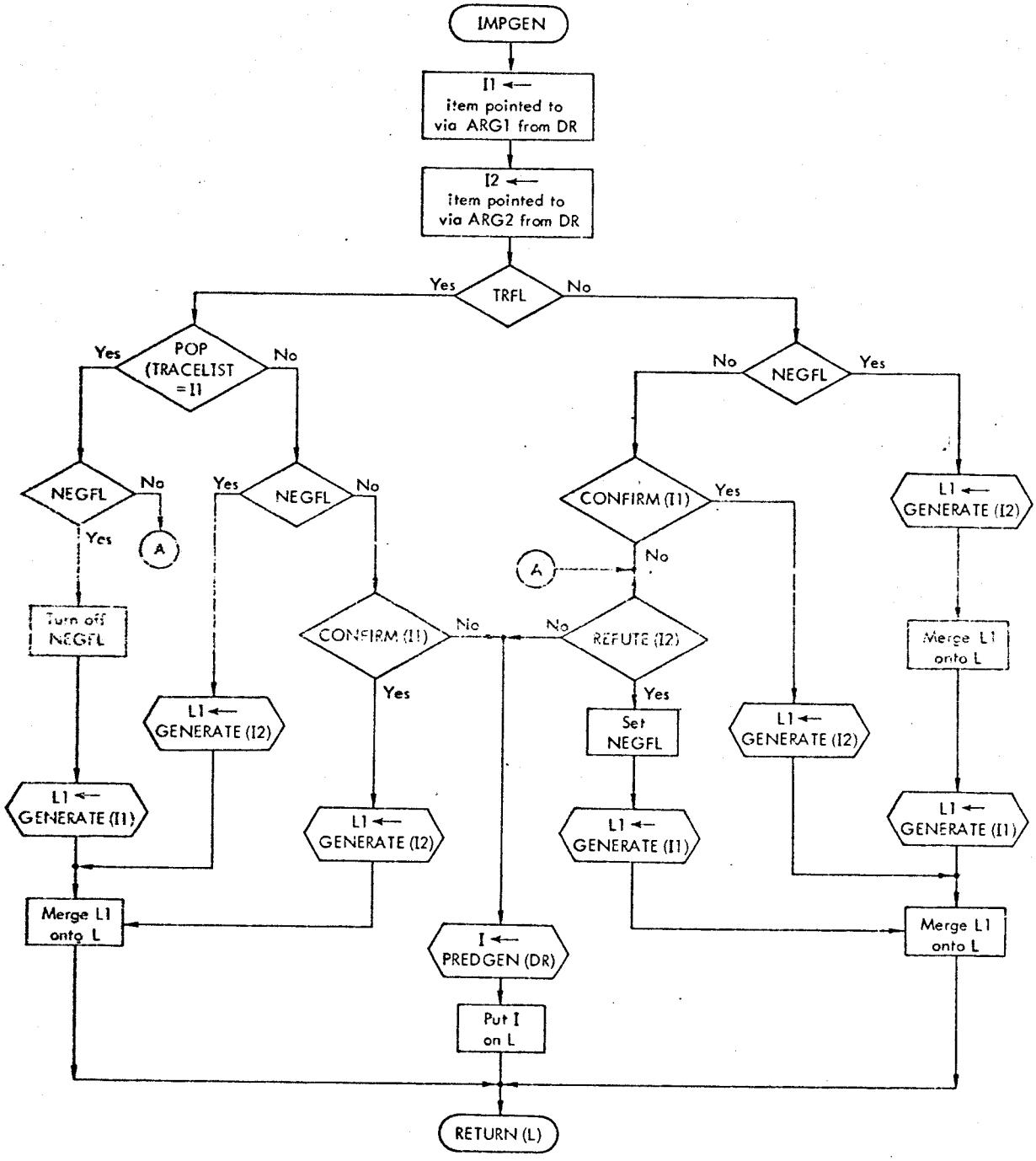

PREDGEN (DR, L1)
NI ← new item
NEGFL
Yes
Negate NI
Turn off NEGFL
No
Any more system relations from DR
No
RETURN (NI)
Yes
R ← next system relation
R = Converse argument, scope or restriction
Yes
No
R = VB
No
Yes
I ← variable pointed to
V ← new variable
Make V the substitution for I
Make NI VB→V
Any more items pointed to via R
No
Yes
I ← next item
I on L1
Yes
No
I a constant
Yes
Make NI point to I via R
No
I a varicble
Yes
I ← substitution for I
No
Add I to the list L1
I ← PREDGEN (I, L1)

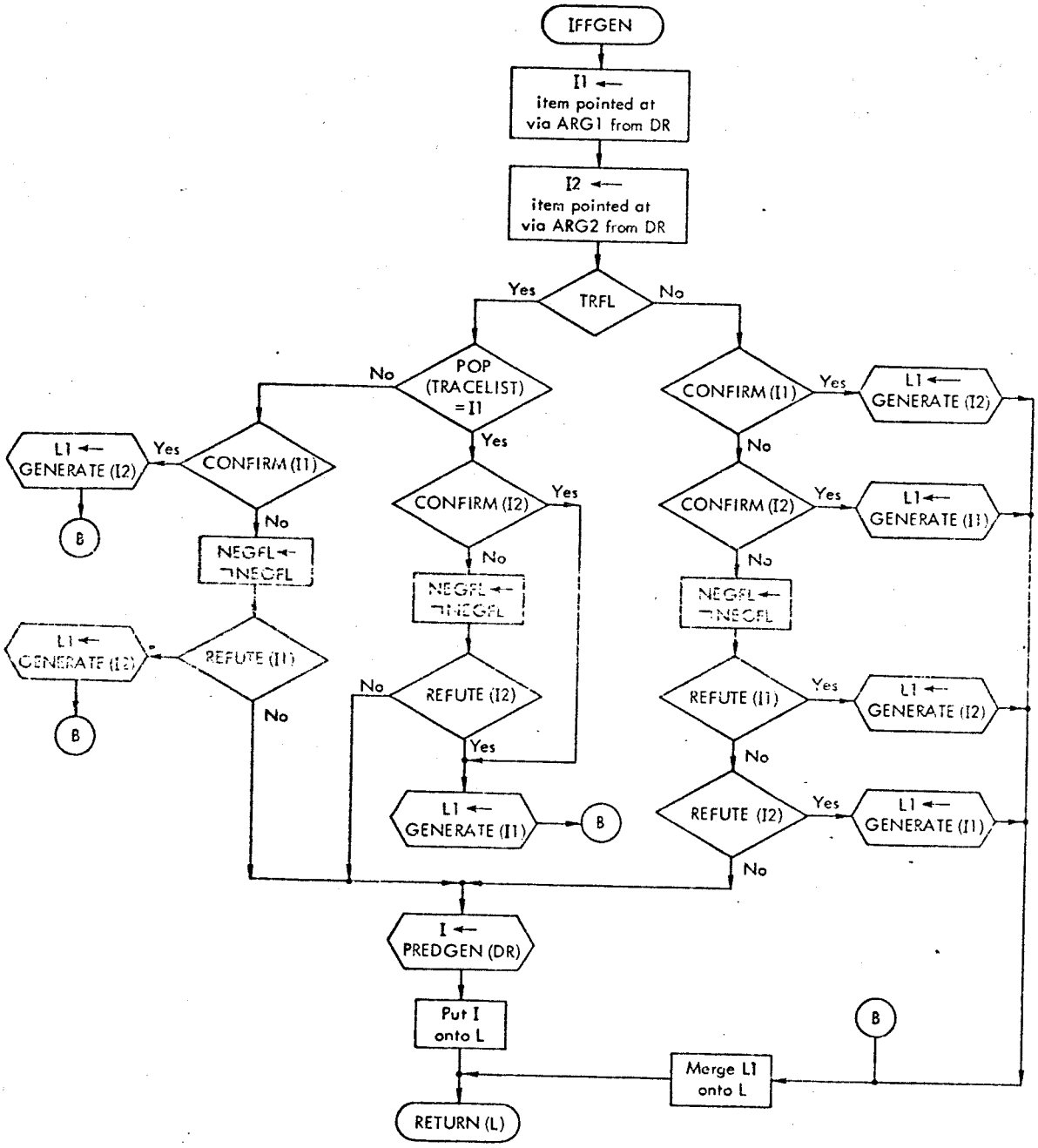

GENERATE (DR)
L ← a new list
DR a constant
Yes
Put DR on L
RETURN (L)
No
TRFL
Yes
No
TRACELIST empty
No
DR Q ∀
Yes
ALLGEN
Yes
No
DR Q ∃
Yes
EXISTSGEN
No
DR OP NOT
Yes
NOTGEN
No
DR OP IMPLIES
Yes
IMPGEN
No
DR OP IFF
Yes
IFFGEN
No
DR OP AND
No
DR OP P
Yes
ORGEN
Yes
ANDGEN
No
DR OP MUTIMP
Yes
MUTGEN
No
Put PREDGEN (DR, (DR)) on L
RETURN (L)

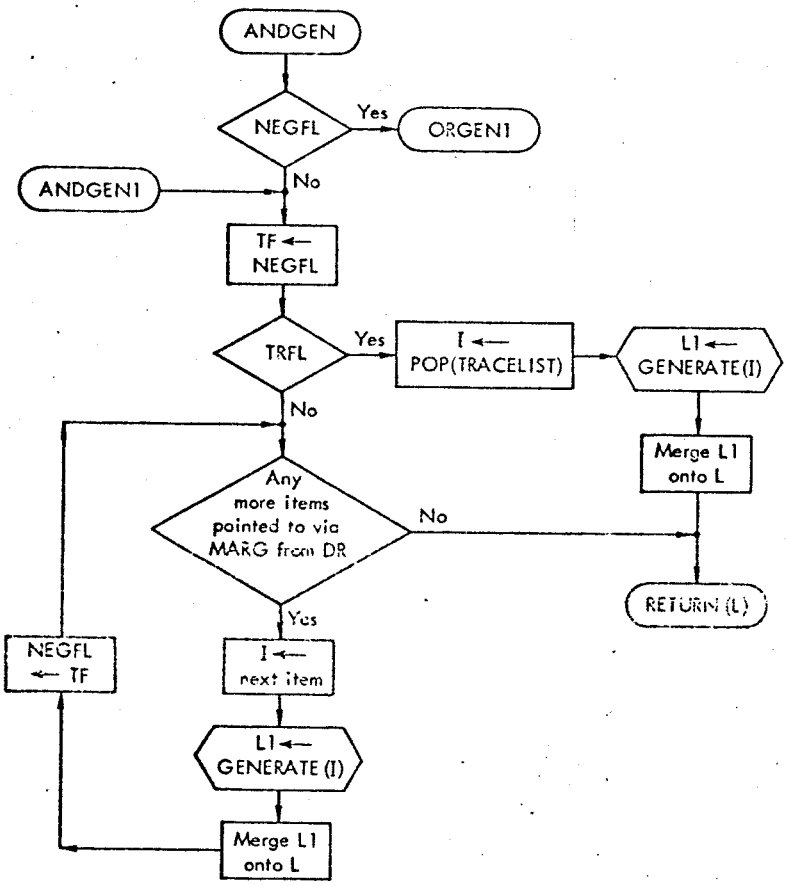

NOTGEN
NEGFL ← ¬NEGFL
TRFL
Yes
POP TRACELIST
No
I ← item pointed at via ARG from DR
L1 ← GENERATE (I)
Merge L1 onto L
RETURN (L)

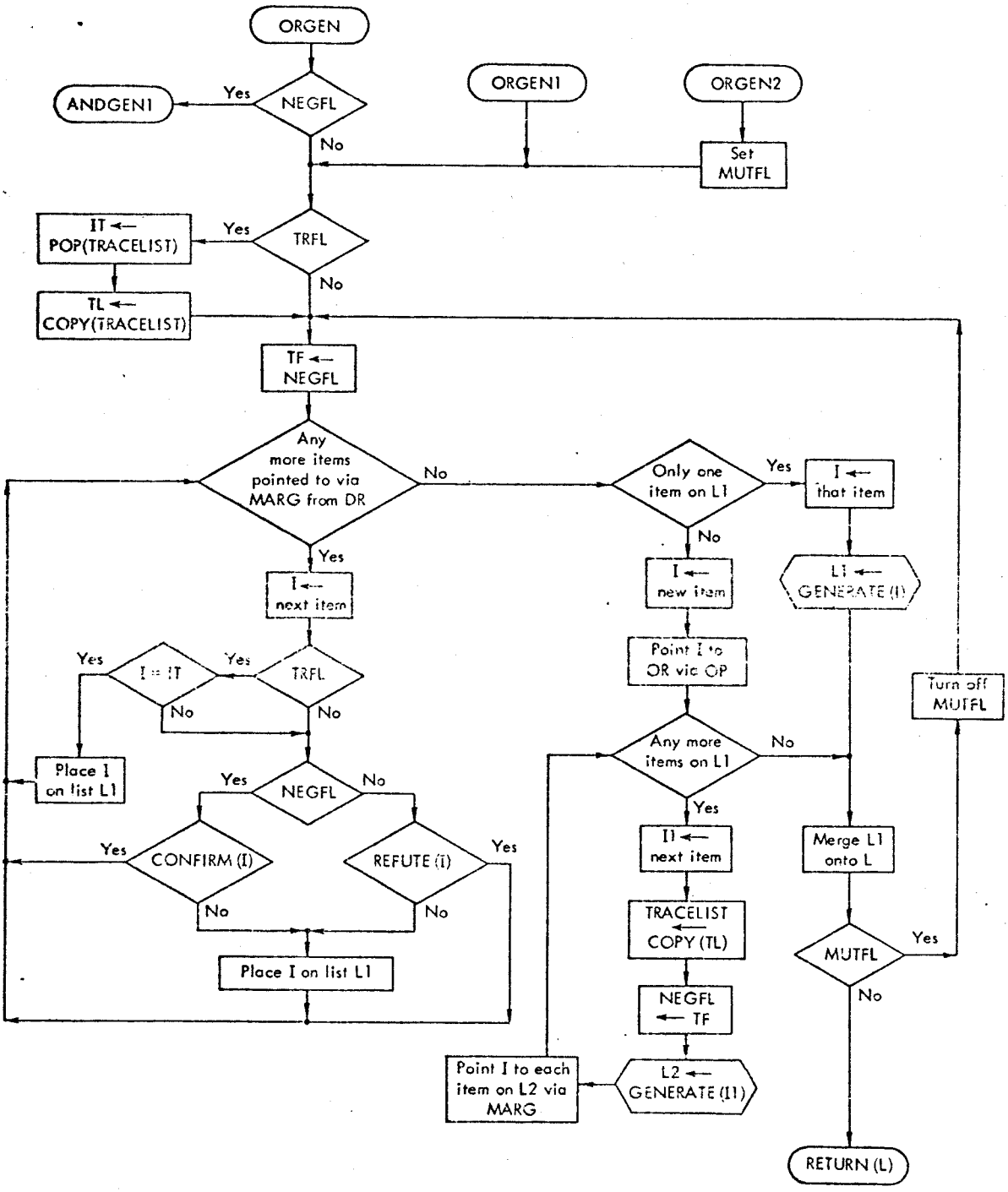

ALLGEN
NEGFL
Yes
EXISTSGEN1
ALLGEN1
No
V ← item pointed to via VB from DR
RI ← item pointed to via R from DR
Is there a list of possible substitutions for V
No
Yes
Any more items in the list of possible substitutions
No
Yes
PS ← next possible substitution
CHECK (PS, V, RI)
F
T
Put PS in the substitution list for V
Substitution list ← FINDSUBS (RI, V)
TF ← NEGFL
TRFL
Yes
POP TRACELIST
No
TL ← COPY(TRACELIST)
Any more items in substitution list
No
RETURN (L)
Yes
VS ← next item on substitution list
SI ← item pointed to via S from DR
Make VS the substitution for V
L1 ← GENERATE (SI)
Merge L1 onto L
NEGFL ← TF
No
TRFL
Yes
TRACELIST ← COPY (TL)

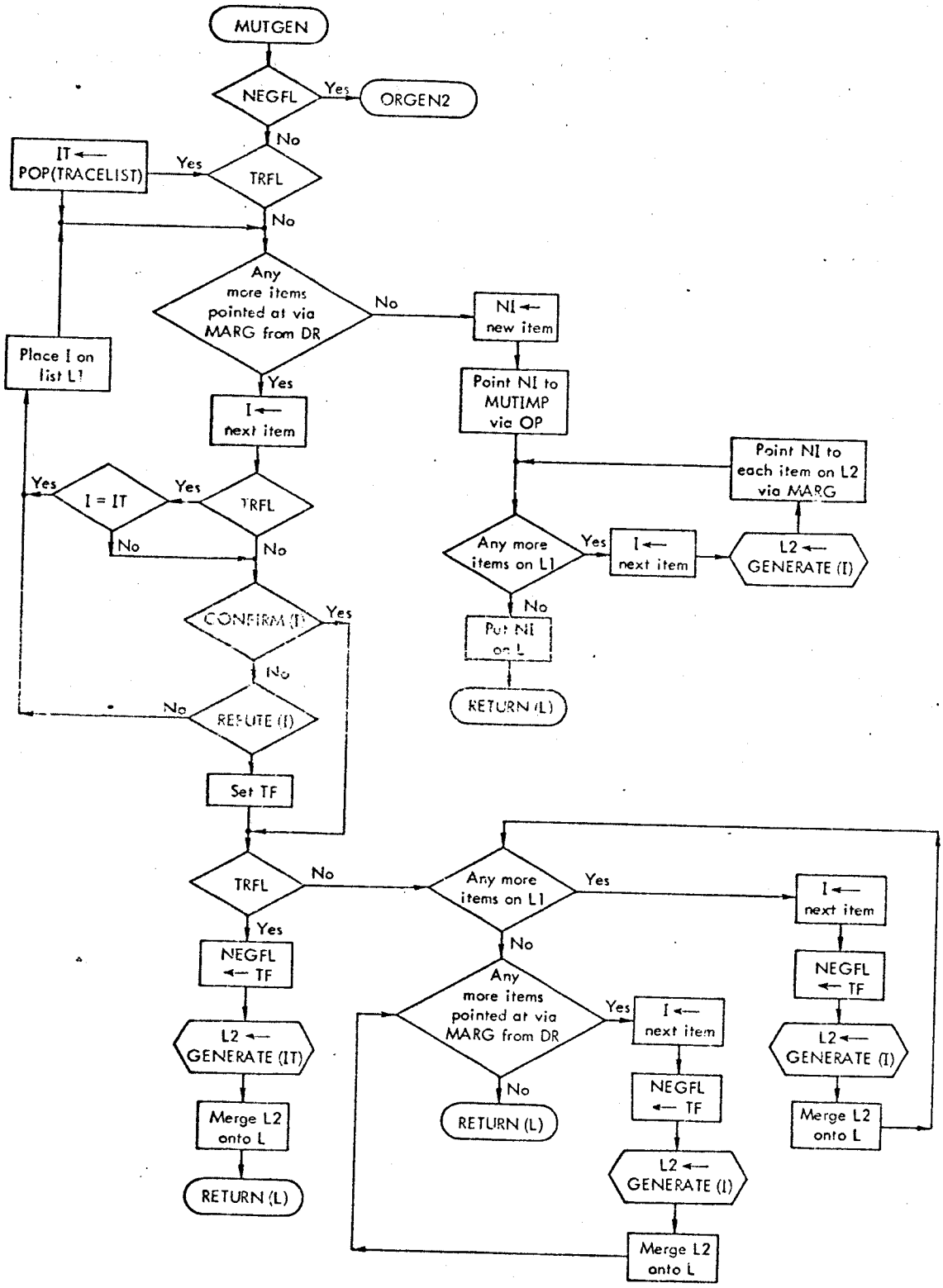

EXISTSGEN
NEGFL
Yes
ALLGEN1
No
VI ← item pointed to via VB from DR
EXISTSGEN1
RI ← item pointed to via R from DR
I ← new constant item
Make I the substitution for V
TF ← NEGFL
TT ← TRFL
Turn off NEGFL and TRFL
Establish as independent statements GENERATE (RI)
SI ← item pointed to via S from DR
NEGFL ← TF
TRFL ← TT
TRFL
Yes
POP TRACELIST
No
L ← GENERATE (SI)
RETURN (L)


IMPGEN
I1 ← item pointed to via ARG1 from DR
I2 ← item pointed to via ARG2 from DR
TRFL
Yes
No
POP (TRACELIST = I1
NEGFL
CONFIRM (I1)
REFUTE (I2)
A
Turn off NEGFL
L1 ← GENERATE (I1)
L1 ← GENERATE (I2)
Set NEGFL
Merge L1 onto L
I ← PREDGEN (DR)
Put I on L
RETURN (L)


IFFGEN
I1 ← item pointed at via ARG1 from DR
I2 ← item pointed at via ARG2 from DR
TRFL
Yes
No
POP (TRACELIST) = I1
No
Yes
CONFIRM (I1)
Yes
No
L1 ← GENERATE (I2)
B
NEGFL ← ¬NEGFL
REFUTE (I1)
No
CONFIRM (I2)
Yes
No
REFUTE (I2)
No
Yes
L1 ← GENERATE (I1)
B
CONFIRM (I1)
Yes
L1 ← GENERATE (I2)
No
CONFIRM (I2)
Yes
L1 ← GENERATE (I1)
No
NEGFL ← ¬NEGFL
REFUTE (I1)
Yes
L1 ← GENERATE (I2)
No
REFUTE (I2)
Yes
L1 ← GENERATE (I1)
No
I ← PREDGEN (DR)
Put I onto L
RETURN (L)
B
Merge L1 onto L


ANDGEN
NEGFL
Yes
ORGEN1
ANDGEN1
No
TF ← NEGFL
TRFL
Yes
I ← POP(TRACELIST)
L1 ← GENERATE(I)
No
Merge L1 onto L
Any more items pointed to via MARG from DR
No
RETURN (L)
Yes
NEGFL ← TF
I ← next item
L1 ← GENERATE (I)
Merge L1 onto L


ORGEN
ANDGEN1
Yes
NEGFL
No
ORGEN1
ORGEN2
Set MUTFL
IT ← POP(TRACELIST)
Yes
TRFL
No
TL ← COPY(TRACELIST)
TF ← NEGFL
Any more items pointed to via MARG from DR
No
Yes
I ← next item
Only one item on L1
Yes
I ← that item
No
I ← new item
L1 ← GENERATE (I)
Point I to OR via OP
Yes
I = IT
Yes
TRFL
No
No
Place I on list L1
Yes
NEGFL
No
Yes
CONFIRM (I)
REFUTE (I)
Yes
No
No
Place I on list L1
Any more items on L1
No
Yes
I1 ← next item
TRACELIST ← COPY (TL)
NEGFL ← TF
L2 ← GENERATE (I1)
Point I to each item on L2 via MARG
Merge L1 onto L
MUTFL
Yes
No
Turn off MUTFL
RETURN (L)


MUTGEN
NEGFL
Yes
ORGEN2
No
IT ← POP(TRACELIST)
Yes
TRFL
No
Any more items pointed at via MARG from DR
No
NI ← new item
Point NI to MUTIMP via OP
Point NI to each item on L2 via MARG
Any more items on L1
Yes
I ← next item
L2 ← GENERATE (I)
No
Put NI on L
RETURN (L)
Place I on list L'
Yes
I ← next item
Yes
I = IT
Yes
TRFL
No
No
CONFIRM (I)
Yes
No
No
REFUTE (I)
Set TF
TRFL
No
Any more items on L1
Yes
I ← next item
NEGFL ← TF
L2 ← GENERATE (I)
Merge L2 onto L
No
Any more items pointed at via MARG from DR
Yes
I ← next item
NEGFL ← TF
L2 ← GENERATE (I)
Merge L2 onto L
No
RETURN (L)
Yes
NEGFL ← TF
L2 ← GENERATE (IT)
Merge L2 onto L
RETURN (L)

References

1. Abelson, R. P., Reich, C. M. Implicational molecules: a method for extracting meaning from input sentences. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, 641-647.

2. Aron, J. D. Information systems in perspective. Computing Surveys 1,4 (Dec., 1969), 213-236.

3. Ausubel, D. P. A cognitive structure view of word and concept meaning. Readings in the Psychology of Cognition, Anderson, R. C. and Ausubel, D. P. (Eds.), Holt, Rinehart and Winston, New York, 1965, 58-75.

4. 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.

5. Bar-Hillel, Y. A demonstration of the nonfeasibility of fully automatic high quality translation. Language and Information Selected Essays on Their Theory and Application, Bar-Hillel, Y., Addison-Wesley, Reading, Mass., 1964, 174-179.

6. Becker, J. D. The modeling of simple analogic and inductive processes in a semantic memory system. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, 655-668.

7. Black, F. A. deductive question-answering system. Semantic Information Processing, Minsky, M. (Ed.), M.I.T. Press, Cambridge, Mass., 1968, 354-402.

8. Brooks, F. P., Jr. and Iverson, K. E. Automatic Data Processing: System/360 Edition, Wiley, New York, 1969.

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

10. _____ Meaning and Necessity A Study in Semantics and Modal Logic. The University of Chicago Press, Chicago & London, 1956.

11. _____ Meaning postulates. In /10/, 222-229. Originally in Phil. Studies 3(1952) 65-73.

12. Cartwright, R. L. Negative existensials. Philosophy and Ordinary Language, Caton, C. E. (Ed.), U. of Illinois, Urbana, 1963, 55-66.

13. Colby, K. M. and Smith, D. C. Dialogues between humans and an artificial belief system. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, 319-324.

14. Cuadra, C. A. (Ed.). Annual Review of Information Science and Technology, V.4. Encyclopedia Britannica, Chicago, 1969.

15. Craig, J. A., Berezner, S. C., Carney, H. C., Longyear, C. R. DEACON: direct English access and control. Proc. FJCC, 1966, 365-380.

16. Darlington, J. L. Theorem provers as question answerers. Unfolioed ms. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, abstract p. 317.

17. DiPaola, R. A. The recursive unsolvability of the decision problem for the class of definite formulas. J. ACM 16,2 (April, 1969), 324-327.

18. Dodd, G. G. Elements of data management systems. Computing Surveys 1,2 (June, 1969), 117-133.

19. Elliott, R. W. A model for a fact retrieval system. Unpublished Ph.D. Dissertation, University of Texas, Austin, Texas, 1965.

20. Feldman, J. A. and Rovner, P. D. An ALGOL-based associative language. Comm. ACM 12,8 (Aug., 1969), 439-449.

21. Fillmore, C. J. The case for case. Universals in Linguistic Theory, Bach, E. and Harms, R. T. (Eds.), Holt, Rinehart and Winston, New York, 1968, 1-88.

22. _____ Lexical entries for verbs. Foundations of Language, 4,4 (Nov., 1968), 373-393.

23. Garfinkel, A. J. Semantic representation theory. TM-3687/000/00, System Development Corporation, Santa Monica, California, Sept., 1967.

24. Green, B. F., Wolf, A. K., Chomsky, C., Langhery, K. Baseball: an automatic question answerer. Computers and Thought, Feigenbaum, E. A. and Feldman, J. (Eds.), McGraw-Hill,, New York, 1963, 207-216.

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

37. _____, MacKay, D. M., Maron, M. E., Scriven, M., Uhr, L. Computers and comprehension. RM-4065-PR, The Rand Corporation, Santa Monica, Calif., April, 1964.

38. Kuhns, J. L. Answering questions by computer: a logical study. RM-5428-PR, The Rand Corporation, Santa Monica, Calif., Dec., 1967.

39. Lakoff, G., Ross, J. R. Is deep structure necessary? Duplicated. Cambridge, Mass.

40. Levien, R. E. Relational data file: experience with a system for propositional data storage and inference execution. RM-5947-PR, The RAND Corporation, Santa Monica, Calif., April, 1969.

41 Lindsay, R. K. Inferential memory as the basis of machines which understand natural language. Computers and Thought, Feigenbaum, E. A. & Feldman, J. (Eds.), McGraw-Hill, New York, 1963, 217-233.

42. Linsky, L. Reference and referents. Philosophy and Ordinary Language, Caton, C. E. (Ed.), U. of Illinois, Urbana, Illinois, 1963, 74-89.

43. Lyons, J. Structural Semantics An Analysis of Part of the Vocabulary of Plato. (Publications of the Philological Society, 20) Basil Blackwell, Oxford, 1963.

44. McCawley, J. D. The role of semantics in a grammar. Universals in Linguistic Theory, Bach, E. and Harms, R. T. (Eds.), Holt, Rinehart and Winston, New York, 1968, 124-169.

45. National Academy of Sciences. Language and Machines, Computers in Translation and Linguistics. NAS, Washington, D. C., 1966.

46. Neisser, U. Cognitive Psychology. Appleton-Century Crofts, New York, 1967.

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

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

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

50. _____ Word and Object, MIT Press, Cambridge, Mass., 1960.

51. _____ From a Logical Point of View, Harper & Row, New York, 1961.

52. Raphael, B. SIR: semantic information retrieval. Semantic Information Processing, Minsky, M. (Ed.), MIT Press, Cambridge, Mass., 1968, 33-145.

53. Reitman, W. R. Cognition and Thought. Wiley, New York, 1965.

54. Reynolds, J. C. A generalized resolution principle based upon context-free grammars. Proc. IFIPS Conf., 1968, H10.

55. Robinson, J. A. A machine-oriented logic based on the resolution principle. J. ACM 12,1 (Jan., 1965), 23-41.

26. Hansen, W. J. Compact list representation: definition, garbage collection, and system implementation. Comm. ACM 12,9 (Sept., 1969), 499-507.

27. Houghton, B. (Ed.). Computer Based Information Retrieval Systems. Archon Books, Hamden, Conn., 1969 (c. 1968).

28. Hunt, E. B., Marin, J., Stone, P. J. Experiments in Induction. Academic Press, New York, 1966.

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

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

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

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

33. Kellogg, C. H. A natural language compiler for on-line data management. Proc. FJCC, 1968, 473-493

34. Klein, S. Automatic paraphrasing in essay format. Mechanical Translation 8,3&4 (June-Oct., 1965), 68-83.

35. Knowlton, K. C. A programmer's description of L^6. Comm. ACM 9,8 (Aug., 1966), 616-625.

36. Kochen, M. (Ed.). The Growth of Knowledge. Wiley, New York, 1967.

56. Sandewall, E. J. LISP-A: a lisp-like system for incremental computing. Proc. SJCC 32(1968), 375-384.

57. Schank, R. C. A conceptual dependency representation for a computer-oriented semantics. Technical Report No. CS130, Computer Science Department, Stanford University, March, 1969.

58. 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.

59. Shapiro, S. C. A memory net structure: present implementation and a proposed language. Technical Report #53, Computer Sciences Department, U. of Wisconsin, Madison, Wisconsin, Dec., 1968.

60. _____ 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.

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

62. Simmons, R. F. Answering English questions by computer: a survey. Comm. ACM 8,1 (Jan., 1965), 53-69.

63. _____ Natural language question-answering systems: 1969. Comm. ACM 13,1 (Jan., 1970), 15-30.

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

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

66. _____, Klein, S., McConlogue, K. Indexing and dependency logic for answering English questions. American Documentation 15,3 (July, 1964), 196-204.

67. Slagle, J. R., Chang, C. L., Lee, R. C. T. Completeness theorems for semantic resolution in consequence-finding. Proc. Int. Jt. Conf. Art. Intel., Walker, D. E. and Norton, L. M. (Eds.), Washington, D. C., 1969, 281-285.

68. Strawson, P. F. On referring. Philosophy and Ordinary Language, Caton, C. E. (Ed.), U. of Illinois, Urbana, Ill., 1963, 162-193.

69. Towster, E. Several methods of concept-formation by computer, unpublished Ph.D. dissertation, University of Wisconsin, Madison, Wisconsin, 1970.

70. Weizenbaum, J. Symmetric list processor. Comm. ACM 6,9 (Sept., 1963), 524-544.

71. _____ ELIZA -- a computer program for the study of natural language communications between man and machine. Comm. ACM 9,1 (Jan., 1966), 36-45.

72. 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., Sept., 1967.