A LANGUAGE AND DATA STRUCTURE FOR FACT RETRIEVAL

by

William Louis Ash

TECHNICAL REPORT #19 - June, 1972

ABSTRACT

A LANGUAGE AND DATA STRUCTURE FOR FACT RETRIEVAL

by
William Louis Ash

Chairman: Edgar H. Sibley

A computer system that results in a useful and quite natural vehicle in which fact retrieval systems can be constructed quite easily is presented and analyzed. The system, called TRAMP, consists of a simulated associative machine providing the storage structure, and a relational language for that associative storage structure providing a question-oriented data structure. The TRAMP system is a computer language with applicability to a class of problems that are best represented as associations between objects, or by a relational data structure. In particular, the language contains potent fact retrieval capabilities in the sense that it automatically deduces implications of facts resident in its store.

We use the term "fact retrieval" here to mean the extraction of facts from a data store regardless of whether or not the information explicitly resides in the store. Thus, fact retrieval includes "document" retrieval—simply locating and extracting a data item from memory (primary or secondary)—but is mainly concerned with strategies for making inferences from data in memory. As a very simple illustration: if we know that A is greater than B, then from that data item we should be able to conclude that B is less than A. To more clearly define the problem it is necessary to place constraints on the deduction. For our purposes these constraints will have to do with time, space, ease of application, and general utility in a realistic environment.

The TRAMP language and system is described and motivated. Comparison of this system to other fact retrieval approaches is discussed. The implementation of the system (on an IBM 360/67) is described. Of perhaps most importance is the algorithm, along with its proof of correctness, for processing the relational language, thereby yielding a very simple manner of handling what has proved to be an elusive problem. The language is critically evaluated, examples given of how it works and how it can be used, a general discussion and critique of the field of fact retrieval, and the user's manual are also included.

FOREWARD

The following is a guide for reading this dissertation. Chapter 1, entitled "Introduction," is an introduction to the subject area with which this dissertation is concerned, rather than an introduction to the dissertation itself.

Chapter 2 discusses the successes and failures of others working in the areas of question-answering and fact retrieval. In so doing, a somewhat terse but complete explication of Robinson's resolution theorem-proving strategy is given. The terminology thus introduced appears without further explanation in the later chapters. The reader may want to refer to pages 22-29 for various definitions of technical terms appearing in the context of theorem-proving and formal logic.

Chapters 3 and 4 deal with the two sub-languages of the TRAMP language. Each chapter attempts to motivate the respective language, define it, and demonstrate its utility. Chapter 3 explains the actual implementation of the associative language in some detail, while the implementation of the relational language is deferred to chapter 6.

Chapter 5 discusses in both general and specific terms how the components of the TRAMP language interact to form the final system. Section 5.1 concerns specific problems of the present implementation and is perhaps not of general interest.

Chapter 6 represents the primary theoretical contribution of this dissertation. Here mathematical graph theory is employed to represent, solve and prove the correctness of that solution for the problems posed by the relational language of Chapter 4. This chapter is actually quite independent of the rest of the dissertation, in that the results are not restricted to the present work. However, the results have been motivated by placing the problem in the context of the TRAMP language, thereby making the reading of chapter 6 difficult without some familiarity with TRAMP (it would be suggested that sections 3.1, 3.2, and 4.1 be prerequisite to understanding chapter 6).

Chapter 7 presents an example of an application of the TRAMP language but, more importantly, the TRAMP language is critically evaluated and compared to other work. Perhaps significantly, chapter 7 also includes a critique of the general area of "question-answering" and questions some of the results and demonstrations that have arisen in that field.

Chapter 8 gives a short summary of the work that was performed as well as of this report of that work.

ACKNOWLEDGMENTS

This work was carried out under the patient supervision of Professor Edgar Sibley, to whom I owe a great debt.

The work was strongly influenced by many fruitful discussions with Professor John Seely Brown, now of the University of California. Indeed, without his encouragement this project might never have been undertaken.

Professor Walter Reitman has my deep gratitude for the strong encouragement and support that he offered.

The services of Professors Alan Merten, Stephen Pollock and Fred Tonge on my doctoral committee are sincerely appreciated.

This research was supported by the Department of Defense, Advanced Research Projects Agency, Contract No. DA-49-083 OSA-3050, and in part by National Institute of Health, Grant No. MHI2977-02, and also by the ISDOS project at the University of Michigan.

TABLE OF CONTENTS

A LANGUAGE AND DATA STRUCTURE FOR FACT RETRIEVAL

Chapter 1: INTRODUCTION

Since the inception of electronic computing machines, and with accelerated growth in recent years, scientists have been investigating the problems of "mechanical question-answering." The area of research termed "question-answering" tends to blend together two quite distinct fields of study: natural language and fact retrieval. There has been considerably more interest and active research into the problems of machine comprehension of natural language (see Simmons [1] for a bibliography of same) than into fact retrieval. The language problem is: to parse mechanically a sentence in a loosely structured natural language in order to extract its information content; the natural language queries must be correctly parsed to retrieve the relevant information previously extracted and stored. Thus, the natural language problem as pertains to question-answering can be thought of as the interface between the question-asker and the question-answerer.

Once this interface exists, or assuming that it did, we have the following diagram (Fig 1.):

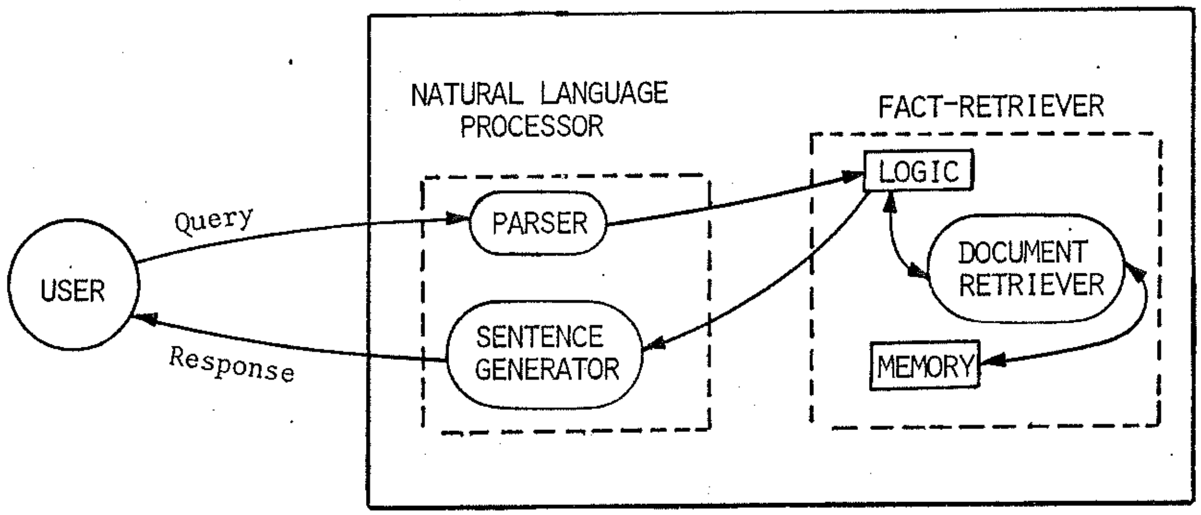


FIGURE 1: A MODEL OF QUESTION-ANSWERING

In this model, the natural language processor is the interface between the user and the fact retriever. The "sentence generator" in the diagram is a relatively simple device that generates English sentences, i.e. translates from the internal form to something intelligible to the user. The "parser" has the (as yet unsolved) problem of translating from an informal, ambiguous, natural language to some *canonical form* intelligible to the fact retriever. The fact retriever itself, then, deals with a highly structured, formal language. Its task is to retrieve *facts*, expressed or implied, from the memory of the system.

We shall be concerned with the above model for user interaction with a question-answering system, but we wish to point out certain reservations about the model. The sharp dichotomy made above between the "Natural Language Processor" and the "Fact Retriever" is quite artificial. This author strongly feels that, in fact, ultimately there is no distinction. The two processes are not only highly interdependent, but at root, the *same* basic (artificial intelligence) problems will have to be solved for either process to be realistically effective. For example, it is widely recognized that effective processing of natural language requires an "efficient" mechanism for making logical inferences. This roughly is just the "Logic" box in Fig. 1. Perhaps more significant is the point of view (held by this author and corroborated by many discussions with colleagues) that when we are able to successfully construct a meaningful processor for a natural language, we will, in fact, have transcended the need for an explicit formal fact retriever.

Reiterating, it is our opinion that the same basic mechanisms are required for construction of either of the large (dotted-line) boxes in the diagram of Fig. 1, and that the distinction made is an artificial one. Nonetheless, given the present state of the art, the model proposed in that diagram is a useful one and a reasonable way to approach the area of question-answering. Later, we will further decompose the "Fact Retriever" box into *syntactic* and *semantic* components, again motivated by the conviction that the decomposition, though perhaps not quite realistic, provides a "handle" on the problem and is a useful approach. (See Green [2] for a somewhat extreme example of this position.)

Before proceeding, the terminology that we shall be using in the sequel requires more precise definition, as it is, unfortunately, not too uncommon in the computer sciences for many persons to use the same words and terms to refer to different things. We shall, following Cooper [3], distinguish between: i) question-answering; ii) information retrieval—or document retrieval; and iii) fact retrieval.

By a *Question-answerer* we shall mean a complete system (such as all of Fig. 1) to which a user poses an English, or English-like, question and from which he receives a response. We shall use the term *fact retrieval* to refer to a system (e.g. the right-hand dotted box in Fig. 1) which is capable of retrieving facts from a memory or data base, regardless of whether or not the fact is explicitly present, or only a consequence of what is explicitly present. *Information* or *document retrieval* is a term we shall use to refer to the retrieval of data items explicitly present in the data base (using methods such as binary search, hash coding, etc. Again refer to Fig. 1).

One other piece of terminology that we shall be using is being popularized by the Data Base Task Group of the CODASYL committee but is still unfamiliar to many people. This is the distinction between a *data structure* and a *storage structure*. We will use the term storage structure to mean the actual representation of information on the physical medium, be it tape, disc, drum, magnetic core, or whatever, and the physical structure of that representation, e.g. linked list. The term data structure will refer to the logical rather than physical representation of data (when the structures are simple the two may coincide). The data structure is the logical structure into which the programmer organizes his data to accomodate and facilitate his way of thinking about his problem. As pertains to TRAMP, the language described in this report, the storage structure is associative, while the data structure is relational. Another example of the distinction is in SIMSCRIPT II [4], where a multi-dimensional array as a data structure is mapped onto a fairly sophisticated storage structure consisting of vectors of pointers to vectors.

The present report is concerned with an approach to the problem of fact retrieval. Referring to the dichotomy in Fig. 1, the fact retriever works with a highly structured, canonical representation of both information and user queries. The fact retriever has the task of effectively and efficiently retrieving and deducing information from the explicit data in its storage medium. What is involved in the process of mechanically retrieving facts? First of all, we must have a base information retrieval system that is fast and efficient. We can expect that any mechanical retrieval system, to be worth implementing on a computer, must necessarily contain a very large amount of information [5]. This information must be appropriately and efficiently compressed so as to minimize utilization of the hardware storage device (typically a disc or drum for a realistically large system), and perhaps more importantly stored in such a way as to minimize retrieval search time and cost. In the proposed system, named TRAMP, a software simulation of an associative memory is the primary strategy for efficient storage and retrieval of data items.

Superimposed on top of the TRAMP associative substructure is a language for performing various logical operation on data, and more importantly, a language for describing how the data can be expanded to include its implications; i.e. a language for describing rules by which valid deductions can be made and information inferred that is not explicitly resident in the store. As a somewhat trivial, but nonetheless realistic example, consider a retrieval system that contains the data item: BOB IS JANE'S HUSBAND. From this data item we would certainly expect the system to *implicitly* "know" that JANE IS BOB'S WIFE. (In general more interesting and fruitful relations will arise in practice, but for illustrative purposes we will restrict ourselves almost entirely to familial relations at the expense of appearing trite.) We would also like the system to "know" various other things that are implied by the statement relating BOB and JANE. For example we would not think the system very bright if knowing that JANE IS BOB'S WIFE, it could not answer: "WHO IS JANE MARRIED TO?" or "IS BOB A BACHELOR?"

Thus, the fact retrieval system must be capable of making direct logical inferences such that if it "knows" that $A \supset B$ and it also "knows" A, then it must be able to apply modus ponens to infer B. It must be able to perform, as well, the deduction that would lead it from BOB IS JANE'S HUSBAND to JANE IS MARRIED TO BOB. We can think of this as a different type of capability, even though it can certainly be phrased and deduced in the propositional calculus. That is every data item will "imply" many other data items which are simply rewordings of the original. For example:

$$\text{BOB IS JANE'S HUSBAND} \supset \left\{ \begin{array}{l} \text{JANE IS BOB'S WIFE} \\ \text{BOB AND JANE ARE MARRIED} \\ \text{JANE IS BOB'S SPOUSE} \\ \text{BOB IS JANE'S SPOUSE} \\ \text{BOB IS MARRIED TO JANE} \\ \text{JANE IS MARRIED TO BOB} \\ \text{etc.} \end{array} \right.$$

But more realistically, we would like some other mechanism (e.g. some type of thesaurus look-up) to transform the information into a standard or canonical form so that the "inference mechanism" need not be overburdened. This remains a very real problem but we would like to relegate it to some other study and, in particular, assume that the "parser" is capable of determining the meaning and intent of a statement and such difficulties as presented by synonyms, word order, phrasing, etc., are not the concern of the fact retrieval mechanism.

The situation for the fact retriever then, is that a "question" is posed to it. Depending on the orientation of the particular system this might take one of the forms: a) Who is John's father? or b) Who is the father of John? or c) Does there exist someone such that he is the father of John (and if so, who is that someone)? and so on. The system then goes into its memory seeking the fact explicitly, or, not finding it, seeking relevant data items that can hopefully be used to deduce the answer. If, in this search, the data item THE FATHER OF JOHN IS DAVID is found, then the system replies "DAVID"; if the data item

JOHN HAS NO PARENTS (along with suitable "axioms" for interpreting this item relative to the particular question) is found, then the system replies negatively. The interesting situation arises in the *maybe* case. Assuming that the system had the ability to fully analyze and interpret familial relations, and it found in its store:

DOROTHY IS JOHN'S GRANDMOTHER

ARNOLD IS DOROTHY'S SON

and no other relevant information, then the true situation might be reflected in the reply:

"THERE IS NOT SUFFICIENT INFORMATION TO ANSWER. HOWEVER, ARNOLD IS *EITHER* JOHN'S FATHER *OR* HIS UNCLE."

Most question-answerers respond with one of three answer *conditions* (most early systems had only these three answers!): Yes, No, Maybe. The "maybe" condition is normally ignored in the sense that no attempt is made to analyze or interpret the facts that have been retrieved in the process of trying to retrieve the desired fact. That is, when the desired fact is not deduced, all that was accumulated along the way (lemmas in the theorem proving framework) are discarded. What is really wanted here is one or more possible answers with associated plausibility indices. Note that this is quite different from simply assigning probabilities to answers as derived from probabilistic rules of inference such as:

WITH PROBABILITY .3, THE SON OF THE GRANDMOTHER IS THE FATHER

WITH PROBABILITY .99, THE SPOUSE OF THE PARENT IS A PARENT, etc.

The problems of fact retrieval, viewed abstractly out of the question-answering context, can be thought of as an attempt at data reduction insofar as we are eliminating redundancies. For example it is clearly redundant to have both JOHN IS THE HUSBAND OF MARY and MARY IS THE WIFE OF JOHN.* If we carry this argument to its logical conclusion we could represent all of the infinitely many theorems of the propositional calculus by storing only the three axioms:

$$(A \supset (B \supset A))$$

$$((A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C)$$

$$((\sim B \supset \sim A) \supset ((\sim B \supset A) \supset B))$$

and providing a mechanism for applying modus ponens and generating proofs. In fact, Green [2] proceeds in just such a formal manner, representing all data as sentences of the predicate calculus (first-order logic) and all retrieval requests as conjectured theorems to be proved. The data base is considered to be a set of axioms to which a proof procedure (resolution [6]) is applied.

The problem, as stated thus far, is not well defined until we stipulate the constraints under which the deductive system is to operate. For example Green has completely solved the "unconstrained problem," i.e. in theory he can infer any and all data implied (syntactically) by other data. However, when we place a few reasonable constraints (such as time and space) on the solution, his system loses a great deal of its theoretical power. Green's work is further discussed in Chapter 2.

*The storage space explosion is naturally much more severe in the case of transitive relations.

We can think of the constraints (normally time and space—or some index of effort expended in making an inference) as determining some inference mechanism such as a formal theorem prover [2] or a relational structure [7,8], or anything else. This mechanism then, will effectively partition (not uniquely!) the total data base into: one set of data items that will be explicit (i.e. reside as data items or "documents" in the memory) and all the rest of the data items which this mechanism can infer from those explicit. In practice we rarely have a partition since the two sets are not disjoint. In that case we call the overlap redundancy. In terms of axiom systems, such as that given above for the propositional calculus, we can formally define redundancy to be the *non-independence* of the axioms (as a function of the rules of inference). The pros and cons of redundant data are discussed in the sequel. The problem of specifying this partition is an interesting one in its own right [9].

It must be stressed that the TRAMP *language* described in this report is not itself a question-answerer, or even a fact retriever. It is a language. But it is a language designed with fact retrieval foremost in mind, and it is a language containing as built-in features many of the requirements of a question-answering system. It will therefore by argued in the sequel that the TRAMP language is in many ways ideal for writing question-answering systems. We will set most of our discussion in the framework of a question-answering system, accordingly. This is because such systems are most concerned with the problems of inferring information from given facts (documents).

Of course all information retrieval can be thought of as question-answering, anyway. Our concern is with general fact retrieval methods but we will often talk of question-answerers since that area readily yields illustrative examples.

In the next three chapters we shall be discussing various approaches and limitations to fact retrieval and in so doing the constraints of operation will come out. We can state for now that the dimensions that should go into any index of expended retrieval effort must include:

1) execution time
2) memory requirements (main & secondary; permanent & scratch)
3) representation of data
4) ease of use
5) flexibility of structure / ease of modification

Chapter 2 presents an evaluation of the various approaches that have been tried thus far. Chapters 3 and 4 deal with the TRAMP approach and its constraints. The later chapters attempt to tie things together discussing implementational details of TRAMP and evaluating its successes and failures, and fitting the TRAMP language into the fact retrieval and question-answering model which we have constructed.

Chapter 2: BACKGROUND

Although there has been a great deal of interest in the general area of "question-answering" (cf. Simmons [1,10]), there has been relatively little concentration on the abstract notion of fact retrieval as defined by the dichotomy in the introduction. "Part of the neglect may ... be due to the well-founded suspicion that the problems of mechanized Fact Retrieval are deep and involved" [3]. However there have been several notable endeavors in this area [2,3,5,8] and we will now survey what has been attempted, how it has succeeded, and what shortcomings remain.

Information retrieval was perhaps first dichotomized (into document- and fact-retrieval) and the term "fact retrieval" first used in the sense that we are using it by Cooper [3]. In this early work, Cooper brought to light the questions with which we shall be concerned, introduced the terminology (which he credits to P. Baxendale of IBM), and established some goals and constraints for fact retrieval. Though his work did not result in a working fact retrieval system of note, or even present a formidable attack on the issues, the problems were well formulated and delineated and hence the work was a significant contribution.

Cooper's aims seemed modest in that he was only interested in being able to answer "yes-no" questions. It is somewhat surprising that question-answering systems were so oriented until relatively recently. Nevertheless, if one has a good grasp on the nature of the problems involved in fact retrieval, one readily sees that restriction to "yes-no" questions in no way simplifies the problem; rather it is a useful abstraction. In particular, Cooper relegated the ability to answer other than "yes-no" questions to the document retrieval system and allowed the fact retrieval system to concentrate on the problems of effective inference. As indicated in Fig. 1, today we tend to think of the document retriever as being an important part of the fact retrieval system and have sufficiently good document retrieval techniques that this is not an encumberance.

Though Cooper strongly distinguished between retrieving facts and documents, he did not decompose the question-answerer as we have. Indeed, his thought was given by:

> "*Proposition 1:* A Fact Retrieval system must normally accept most of its information to be stored, and also its queries, in the form of natural language sentences (e.g. English) rather than in some artificial language selected for the purpose." [3]

He proposes a highly restricted subset of English (generated by an "immediate constituent" grammar) for his language. While there is not enough information available to confidently evaluate his chosen subset, it appears to be one of those "English-like" languages that is very easy to *read*, but very difficult to *write* (since the restrictions pose no problem to the reader but are a serious nuisance to the falsely confident writer who is misled by the seeming lack of formality).

Cooper's basic approach to retrieving facts is essentially a theorem-proving approach. His "Fact Retrieval Algorithm" is the following:

(1) Search for a subset of the set of stored sentences such that the query sentence is a logical consequence of the subset. If found, SUCCESS EXIT, else go to (2).

(2) Search for a subset of the set of stored sentences such that the subset is logically inconsistent with the query sentence. If found, FAILURE EXIT, else go to (3).

(3) Exit—unable to answer.

His approach to theorem proving was perhaps ill-defined (resolution [6] had not yet been discovered) and was important only in that he proposed "a theorem prover" to realize his algorithm. To this end he did formulate a "logical language" in set-theoretic notation employing Aristotelian logic.

Cooper thus proposed a theorem proving attack on fact retrieval. Unfortunately, at that time he did not have available a potent approach to theorem proving, and hence his fact retrieval system bogged down. A few years after Cooper the resolution and unification theorems [6] were discovered and proved yielding the resolution strategy the most acceptable and feasible proof procedure today. With resolution Green [2] was able to implement Cooper's "Fact Retrieval Algorithm." We will return to Green and the resolution principle shortly and examine the result of this implementation.

At the RAND Corp. Levien and Maron have been working for some time on the problem of fact retrieval as it is manifested in the context of library automation [5,11-13]. Although they have approached the computer to begin implementation of their ideas, it is this author's impression that the significance of their work has been, as with Cooper, in setting forth goals and constraints, and the formulation of a design philosophy. From personal communications with persons acquainted with their project, we understand that implementation is under way. Yet from the published material one discerns only that they have given the problem a good deal of thought and come up with a way to approach it. The approach that they formulate, by specifying a fact retrieval language, is a language which in fact closely resembles the TRAMP relational language. However the TRAMP language is currently operational and a prototype version of it was usable as far back as the summer of 1967, whereas Levien and Maron give little hint (in the literature at least) of how they intend to cope with the problems posed by their language.

Their contribution is quite significant however, since, like Cooper, they have given the design philosophy much thought and greatly helped to clarify the problem, its goals and constraints. They have gone beyond Cooper in that their fact retrieval language is a concrete proposal well within reach and one which is well thought out and justified.

A system that externally appears to be similar to TRAMP (but internally bears no resemblance whatever) was proposed in a dissertation by Elliott [8]. His system, called GRAIS (implemented on the CDC 1604 using the SLIP language) stored information as binary relations represented as edges of a graph. Inference was performed by providing nine properties that could be associated with each relation. These properties, e.g. transitive, irreflexive, etc., specified ways in which paths through the graph representation of the data could be traced to retrieve implicit information. As stated above, GRAIS and TRAMP appear to be similar approachs on the surface (we shall, in fact, later be using quite the same terminology to describe TRAMP, i.e. relations thought of as a graph or network [14]), however their realizations were quite different. Furthermore TRAMP has gone beyond GRAIS to the extent of having incorporated virtually all of the extensions which Elliott *proposed* for GRAIS in the "future." Again, it must be kept in mind that unlike any of the systems reviewed here, TRAMP is a *language* and hence a vehicle for fact retrieval, rather than a fact retrieval system.

In formulating his problem, Elliott suggested seven criteria for a question-answerer:

1) Comprehensible input-output format
2) Storage efficiency
3) Inference capability
4) Generality of subject matter
5) Processing time independent of size of data base
6) Revision facility
7) Ability to note inconsistencies in data base.

As we shall see, items (2)-(6) are built-in features of the TRAMP language, while items (1) and (7) are easily programmed in the language.

Let us examine these criteria in detail. The first, comprehensible format, is just Cooper's Proposition 1. The TRAMP language is imbedded in TRAC [15] which is a very elegant "text-processing" language, greatly facilitating the achievement of this first criterion by any fact retrieval system written in TRAMP. Elliott provides as his solution the language of relations. While the language of relations is concise, rich, and perhaps elegant, it is subject to strong criticism when we claim that it is comprehensible: indeed, "comprehensible" is at best an inappropriate adjective. When we discuss Green, shortly, we shall point out some problems with the use of the predicate calculus. Relational languages are markedly similar to the predicate calculus and certainly suffer from the same problems which we shall point out below. For the moment it will suffice to say that these problems are concerned with syntactic rigidity and the fact that it is the very conciseness and richness that yields a language that is perhaps too powerful (in that it is difficult to control and easily gets out of hand to express things that were not intended). We say this because of familiarity with criticism of Green as well as our own experience with the TRAMP relational language. It should of course be pointed out that TRAMP itself is one of those languages that is "impossible to read" (though quite easy to *write*), but the complex and unnatural constructions that render it unreadable are normally generated internally so as to not significantly hamper the programmer and be almost completely transparent to the user of any system programmed in TRAMP.

The second criterion, storage efficiency, is somewhat vague in that it is quite open to interpretation. The question could be effectively argued either pro or con for both GRAIS and TRAMP. However, it will be conceded at the outset that the associative storage scheme used by TRAMP is not very efficient. Rather that scheme was intended to maximize retrieval efficiency and, in fact, was specifically designed to answer criterion (5). But this criterion of efficiency is really too vague to dwell on. Do we speak of efficiency of time or space? What of the trade-off? Is it efficiently stored so that it may be efficiently retrieved? The associative storage mechanism is quite efficient with respect to time, yet is relatively wasteful of storage. GRAIS used the SLIP language [16] for its storage and data structures which could not have been extremely fast; and the doubly linked lists hardly represent efficient storage utilization.

The third criterion, inference capability, we view as being the central problem area, hence redundant. Criteria (4) and (6) seem also to be redundant since one has no *system* without having met them. Item (4) is met by TRAMP implicitly, since it is a language and therefore not capable of being dependent on subject matter. The language certainly has facilities (see Appendix) for revision and deletion, thereby meeting item (6).

The fifth criterion, processing time, was the original problem which the TRAMP associative storage structure was designed to meet. We know of no software retrieval scheme which solves this problem as well. Software associativity involves a minor amount of overhead used in its realization. But this overhead is incurred at virtually the exact same expense regardless of how much data is being searched, i.e. how many data items are stored in the medium in which the desired data item is held.

The last criterion, which concerns inconsistencies in the data base, is something that is not applicable to TRAMP because it is a language. This can certainly be programmed into a system using the TRAMP language, however. As regards a fact retrieval system, this criterion may very well be superfluous. If we accept the very general algorithm put forth by Cooper, we see that whether or not an explicit theorem proof procedure is employed, what we are actually doing is attempting to draw logical conclusions from antecedents and, therefore, in fact we are proving theorems however informally. Formally we know that if our axioms (antecedents) are inconsistent, *any* formula logically follows from them and thus can be proven as a theorem. Viewed in this light, we see that it is superfluous to require that the data items of a fact retrieval system be consistent. Since TRAMP is a language that has found applicability in a wide range of areas not even remotely connected with fact retrieval, it was deemed prudent that such features as consistency be left to the programmer rather than incorporated into the language itself.

Elliott's primary approach to (what we have called) the central problem, i.e. the inference mechanism, is his graph representation and mathematical properties assigned to the edges of the graph. He provides a language for assigning properties to relations (edges) and a relational language for "combining" new relations from previously defined relations, e.g. COMBINE Father with Mother to define Parent. His relational language has several shortcomings which he points out, such as lack of recursiveness and various restrictions on the COMBINE operator. Virtually all of these restrictions on the relational language have been overcome in the TRAMP language (described in Chapter 4).

In summary, Elliott's dissertation was an early, strong approach to a fact retrieval system, which, aside from superficial similarities, is aligned with TRAMP in that the central strategy was to formulate the proper data structure as contrasted with the strategy of Green which was a pure syntactic formalism.

Probably the most significant results in question-answering to date are those of Green [2]. He has taken a purely formal theorem-proving approach to the problem. The information (data base that is being interrogated) is treated as a set of axioms and the question to be answered is posed as a conjecture to be proved as a theorem. A modification of Robinson's resolution proof procedure [6] is employed, yielding constructive proofs whenever the sentence is a theorem of the system. As the resolution theorem assures us of a complete logic system, Green has in principle solved the problem: almost any question can be phrased in the predicate calculus and any valid theorem will be proved using resolution. Unfortunately, in practice things are not so ideal.

Green's most recent system, called QA3, represents an extreme example of the model we have chosen for question-answering (Fig. 1). He tacitly assumes the existence of some interface to transform all communication from the outside into the predicate calculus. The entire data base is thus represented as a set of formulae of first-order logic. The query must also be so formulated. Then the resolution proof procedure is applied to determine whether or not the formula representing the query is a logical consequence of the set of formulae representing the data base (the axioms). Since we know that there can be no decision procedure for the predicate calculus, the completeness of resolution suffices—in theory. In fact, however, there are two major drawbacks to the use of QA3 as a real life question-answerer (even though QA3 is currently being used by the robot at the Stanford Research Institute and elsewhere) as we shall demonstrate.

An example is now in order. Following is a very trivial dialogue with QA3 that demonstrates both the format of information and, more importantly, the manner of representation. We begin by entering two pieces of information into the system:

STATEMENT: MAN(Smith)

STATEMENT: $(x)[\mathrm{MAN}(x) \supset \mathrm{ANIMAL}(x)]$

This says that "Smith is a man," and that "man is an animal," or more precisely, "If x is a man then x is an animal." We now ask "Who is an animal?", which must be phrased "Does there exist a y such that y is an animal? If so, exhibit such a y." and must be asked by typing:

QUESTION: $(\exists y)\mathrm{ANIMAL}(y)$

and the system responds

ANSWER: YES, y = SMITH

The YES answer is given since the conjecture $(\exists y)\mathrm{ANIMAL}(y)$ was proved. The further information "y = SMITH" is provided since "Smith" is an instance of y satisfying ANIMAL(y), or in other words, ANIMAL(Smith) is a theorem.

At this point we shall digress from our discussion of question-answering to examine more closely Robinson's resolution strategy. This digression is felt to be germane to the discussion because only through familiarity with resolution can one appreciate the power of QA3, and also appreciate why that power loses its effectiveness in practice. Above we stated that there were two major drawbacks to QA3. The first has to do with resolution; the second is the manner of representation. Let us now look at the resolution proof procedure.

We must begin by defining our vocabulary. The predicate calculus deals with strings of symbols called *well formed formulas* (wffs). Wffs are composed of *variables* (the letters x,y,z,u,v,w with subscripts); *function letters* (the letters f,g,h,a,b subscripted and also with superscripts to denote the number of arguments); *predicate letters* (P,Q,R,F, G,H sub and superscripted as are function letters); the logical *connectives* $\wedge$ (and), $\vee$ (or), ~ (not); and the existential and universal *quantifiers*. A function of no variables (superscript zero) is a *constant*; a predicate of no variables is a *proposition*. A *term* is a constant, a variable or a function letter f_i^n preceeding n terms. An *atomic formula* is a predicate letter P_i^n preceeding n terms. A *wff* is an atomic formula, the negation of a wff, two wffs joined by a connective, or a wff preceeded by a quantifier. (Sub and superscripts are almost always omitted when no confusion can result and parentheses delimit the arguments to a function or predicate.)

An *interpretation* of a wff S is the selection of a non-empty domain D, the assignment of each function letter f_i^n in S to a mapping $D^n \rightarrow D$ (zero-ary functions, or constants, are assigned to elements of D), and the assignment of each predicate letter P_i^n in S to a mapping $D^n \rightarrow \{T,F\}$. A variable then ranges over the domain D. An interpretation is said to *satisfy* a wff S if S is made to be TRUE under that interpretation, in which case the interpretation is called a *model* for S. A wff is *satisfiable* if there exists a model for it. A wff is logically *valid* if every interpretation is a model for it, or equivalently, if its negation is *unsatisfiable*.

In first-order logic we require that variables represent terms only—a variable can range only over the domain of interpretation, not e.g. predicate letters. A wff is a *closed* formula just in case it contains no *free* variables, i.e. variables not *bound* by (in the scope of) a quantifier. In mechanical theorem-proving formulas are converted to a standard quantifier-free form. A wff S can be algorithmically converted to *prenex conjunctive normal form* S', where all quantifiers occur at the beginning of S' (called the *prefix*), and the rest of S' is an AND of OR's of *literals* (atomic formulas or the negation of atomic formulas) called the *matrix*. Each quantifier ranges over the entire matrix. Each existentially quantified variable is then replaced by a *Skolem function* of those universally quantified variables that preceed it in the prefix (within whose scope it lies). As an example

$$(\exists z)(x)(\exists y)[P(x,y) \supset Q(x,z)]$$

already in prenex form, would be converted to

$$(x)\ [\sim P(x,f(x)) \vee Q(x,a)]$$

where f(x) is the skolem function replacing y, and the constant a is the skolem function of no variables replacing z. In this standard form all variables are universally quantified, so the prefix may be dropped. The matrix is an AND of OR's so it may be represented as a set of *clauses* where each conjunct is a clause, and each clause is an (unordered) set of literals. Thus the canonical form of a wff for resolution is simply a set of clauses, where each clause is a set of literals.

The mechanical conversion of S to S' does not preserve equivalence. However interprovability is maintained so that S is a theorem if and only if S' is a theorem, or more to the point, S is satisfiable if and only if S' is satisfiable.

A proof procedure is said to be *sound* if every theorem of that procedure is a valid formula; it is said to be *complete* if every valid formula is a theorem; consequently a procedure that is both sound and complete has the very desirable property that the theorems of that system exactly coincide with the valid formulas.

Resolution is a "Herbrand refutation" proof procedure. This means that when we try to prove that A is a logical consequence of B, we do so by negating A and attempting to derive a contradiction. This is a very common strategy for proving theorems in everyday mathematics and follows from the simple observation:

$$(B \supset A) \iff (\sim B \vee A) \iff \sim(B \wedge \sim A)$$

Thus we assume $(B \wedge \sim A)$ and attempt to deduce an explicit contradiction thereby showing that no interpretation can satisfy $(B \wedge \sim A)$. We can then conclude that $\sim(B \wedge \sim A)$, or equivalently $(B \supset A)$. Robinson showed that the resolution procedure produces a contradiction from $(B \wedge \sim A)$ if and only if $(B \wedge \sim A)$ is unsatisfiable ($(B \supset A)$ is logically valid), meaning that resolution is a sound and complete proof procedure.

The basic idea of resolution is best illustrated using the propositional calculus (leading to the *ground resolution theorem*). The basic strategy in ground resolution* is very close to the operation of finding prime implicants in switching theory. The validity of the ground resolution principle can be seen from the following tautology (where α and ω are any formulas):

$$[(\sim A \vee \alpha) \wedge (A \vee \omega)] \supset (\alpha \vee \omega) \qquad (1)$$

or equivalently

$$\sim(\alpha \vee \omega) \supset \sim[(\sim A \vee \alpha) \wedge (A \vee \omega)] \qquad (2)$$

In a refutation procedure we are trying to show that a set of formulae (viz. the set of axioms and the negation of the conjectured theorem) can*not* be satisfied. Thus if we were attempting to show that the premise of implication (1) could not be satisfied, we would know, by implication (2) that it suffices to show that the consequent of (1) cannot be satisfied. Since in resolution we are always working with a formula in conjunctive normal form, if any of the conjuncts is unsatisfiable, the entire

*Ground resolution is just resolution applied to formulae in which no variables appear. A predicate of no variables is a proposition. Hence ground resolution is resolution applied to the propositional calculus.

formula is unsatisfiable. This means that if in our set of clauses appear two clauses such as the premise of (1) (called a *complementary pair*), we may add to our set of clauses the consequent $(\alpha \vee \omega)$. In other words, from the two clauses $(\sim A \vee \alpha)$ and $(A \vee \omega)$ we may *infer* the clause $(\alpha \vee \omega)$. This strategy succeeds in deducing an explicit contradiction whenever the empty clause is inferred (since of course the empty clause can only be inferred by resolving the two singleton clauses $(\sim A)$ and (A) which represent the contradiction $(\sim A \wedge A)$).

We define the *resolution space* of a set S of clauses as:

$$R^0(S) = S$$
$$R^{n+1}(S) = R(R^n(S))$$

where R(S), called the resolution of S, is the set of all clauses which are members of S or resolvents (inferred from a complementary pair) of members of S. Thus the resolution procedure generates at each level n, the nth resolution R^n. One of three things must happen: for some n, $R^n = R^{n+1}$ indicating that the set of clauses S is satisfiable; for some n, R^n contains the empty clause indicating that S is unsatisfiable; or else the procedure does not terminate.

The strategy outlined above is not very exciting when we speak of it in terms of the propositional calculus. Indeed, far more efficient methods are known. What Robinson did in [6] was provide the *unification* algorithm (and the proof of its correctness) which in turn provided a means for extending this strategy to first-order logic. Furthermore, his resolution theorem guarantees us that using only the resolution rule of inference we have a complete logic.

The extension to predicate calculus is anything but obvious. The whole idea behind all of the "Herbrand-type" refutation procedures is to demonstrate that the formula (in our case a set of clauses) is not satisfiable under any interpretation. Obviously one cannot enumerate all possible domains and interpretations on domains. However Herbrand proved (about 1930) that it is sufficient to show the unsatisfiability of only a finite subset of the *Herbrand universe*. The Herbrand universe can be thought of as an (infinite) enumeration of all possible interpretations. Technically, the Herbrand universe is simply the set of all terms that can be generated by instantiating terms for variables in the set of clauses. Formally, the Herbrand universe of a set of clauses S, H(S), is defined by:

i) all constants in S are in H(S). If none, $f_1^0 \in H(S)$

ii) for all n-tuples $a_1,\ldots,a_n$, where $a_i \in H(S)$, and all function letters f_i^n in S, the term $f_i^n\ a_1\ \ldots\ a_n$ is in H(S)

iii) nothing else is in H(S).

Except for some relatively rare and uninteresting cases, H(S) is infinite. We have:

HERBRAND'S THEOREM: If S is any finite set of clauses and H its Herbrand universe, then S is unsatisfiable if and only if some finite subset of H(S) is unsatisfiable.

The resolution principle is actually a heuristic for finding that finite subset of H(S). Of course in this search the accrued subset is obviously finite at all times and one never knows whether or not to stop if a contradiction has not been found yet. This is just what we must expect, for by Church's theorem we know that we cannot have a decision procedure (guaranteed to terminate) for first-order logic. As manifested in the resolution procedure, when one infers the empty clause he is finished; or when, at the ith level of resolution, nothing new is inferred he is finished; otherwise it is undecidable whether or not to keep looking. (Note that ground resolution always terminates since no new terms are ever generated and there are only a finite number of resolvents of a finite set of clauses.)

The main difficulty in applying the resolution principle to predicate calculus is being able to identify complementary pairs. For example resolution will readily infer from the pair of clauses

$C_1 = \{Q(x,g(x),y,h(x,y),z,k(x,y,z))\ ,\ P(x,y)\}$

$C_2 = \{\sim Q(u,v,e(v),w,f(v,w),x)\}$

the resolvent clause

$C_3 = \{P(y_1,e(g(y_1)))\}$

The unification algorithm gives an organized method of instantiating terms for variables so that the resolvent is well-defined. An example of another difficulty that unification overcomes is illustrated by the set of clauses:

$S = \{(F(x),F(y)),\ (\sim F(x),F(z)),\ (F(x),\sim F(y)),\ (\sim F(x),\sim F(y))\}$

S is certainly unsatisfiable since an instance of the first clause is the negation of an instance of the fourth clause. Unification presents a mechanical method of getting around the pitfall of resolving the first clause with the second to form the "resolvent" $(F(y),F(z))$ which is just an instance (change of variable) of the first clause and not a true resolvent because it adds nothing new to the set S (similarly for any other pair in S). Specifically, what unification would accomplish is the substitution of x for y in the first and last clauses of S thereby collapsing (unifying) each to a singleton.*

Now let us look at some of the ramifications of the above formality when applied verbatim to question-answering, as was done in QA3. Though resolution is an elegant theory and yields the most powerful method of mechanically proving theorems known today, we still do not have a practicable theorem-prover. While resolution is indeed a powerful heuristic for cutting down the explosion of the Herbrand universe (and a miriad of other techniques and heuristics have since refined resolution), the Herbrand universe is still infinite and still grows at an alarming rate. The resolution theorem guarantees us that if the set of clauses is unsatisfiable, the empty clause will be inferred—in a finite number of iterations. This means that given *enough time and space* a valid formula will be proved. In practice, however, the amounts of time and space required are enormous.

A fine illustration of the data representation problem where predicate calculus is used as the language is provided in the following example taken from [2]. QA3 is given the following facts (shown in LISP notation as they would actually be input):

*In actual implementations of resolution, the unification is performed in two seperate steps requiring a second rule of inference called *factoring*.

```
(IN JOHN BOY)
(FA(X)(IF(IN X BOY)(IN X PERSON)))
(FA(X)(IF(IN X PERSON)(IN X HUMAN)))
(FA(X)(IF(IN X HUMAN)(HAS X ARM 2)))
(FA(Y)(IF(IN Y ARM)(HAS Y HAND 1)))
```

I.e. John is a boy; a boy is a person; a person is a human; a human has two arms; and an arm has one hand. Now the question is asked "How many hands does John have?" This is posed as the conjectured theorem: (EX(X)(HAS JOHN HAND X)). But these five axioms were insufficient to answer the question! The system responded with the comment: "NO PROOF FOUND." The axiom required to answer the question was:

```
(FA(X Y Z M N)(IF(AND(HAS X Y M)(FA(U)(IF(IN U Y)(HAS U Z N))
                                        ))(HAS X Z (TIMES M N))))
```

It is neither obvious why a sixth axiom was necessary, nor what it is that this strange looking axiom says. Essentially, the axiom expresses the heredity property of the predicate HAS (and the necessary arithmetic). Note that this axiom is certainly necessary since the heredity property is not syntactically logical, as evidenced by the counter-example: I HAVE A DOG; MY DOG HAS A TAIL; THEREFORE I HAVE A TAIL (and after having added the sixth axiom such an absurdity is a valid syllogism!). The point to be made is how easy it is to omit pertinent information and how difficult it is to phrase that information.

Another major difficulty with this approach is the syntactic rigidity of the system. Take, for example, the representation of the fact that GEORGE IS AT HOME. This would be translated into the axiom "AT(George,home)." Despite the superficial resemblance, this axiom contains less information than the original English sentence. First, we lose the semantic information that the "home" where George is is George's home. Second, temporal information is disregarded. The problems thus created can be seen if we add the axiom "AT(Jill,home)" from which we can now conclude (probably falsely) that George and Jill are at the same place: $((\exists x)[AT(George,x) \& AT(Jill,x)])$. It is further quite likely that tomorrow, or sometime soon, George will not be at home, yet this would lead to the logical falsehood of the conjunction: $[AT(George,home) \& \sim AT(George,home)]$.

Further represenatational problems are illustrated by the earlier example of "MAN(Smith)." The first axiom was represented in a straightforward manner, leading one to presume that the second could be written "ANIMAL(MAN)" instead of as the implication "$(x)[MAN(x) \supset ANIMAL(x)]$." But of course this would transcend first-order logic and such a straightforward representation would not yield the desired syllogism. This is summed up in [17]:

> "In short, the present lack of a developed theory for representation of semantic content—and therefore the present lack of systematic procedures to translate from natural language to such a representation—preclude the use of first-order theorem-proving procedures to answer questions in all but artificially contrived and controlled environments so restricted ..."

Thus, while Green's application of theorem-proving techniques to question-answering* is quite powerful and elegant, overall, it does not

*Of perhaps even more interest was Green's application of resolution methods to general problem solving by State-Transformation techniques.

provide the desired solution to how one might implement Cooper's fact retrieval algorithm.

Chapter 3: ASSOCIATIVE MEMORIES

The basic "document retrieval" technique used in the TRAMP language is provided by the software simulation of an associative memory. The earliest efforts in this direction (that we are aware of) and the work that provided the biggest stimulus to the present project, was the AL language of Feldman [18]. At virtually the same time that TRAMP was being developed and the first prototype of it became available, Feldman was extending AL into the LEAP language [19]. In one further iteration, on different hardware, LEAP has evolved into SAIL [20]. Although SAIL and TRAMP are both derivatives of AL, the remaining similarities are negligible.

The key idea motivating associative memories is the desire for a *content-addressable* computer memory: an associative memory can effectively be employed to approximate content-addressability. By an associative memory we mean a memory that stores information in ordered n-tuples, called associations, which can be referenced by specifying any of the components of the association. We refer to an association by its contents (components), rather than by any address; indeed, it is the lack of explicit addresses that characterizes an associative machine. The term content-addressable becomes clearer when we see that we reference an association by its contents. More precisely, by a content-addressable memory, we essentially mean one in which the name of a datum contains a dynamic cue to the relevant information about that datum. Content-addressability obviates table lookups, binary search, etc. An associative processor provides a useful approximation to content-addressability.

In recent years the need for this type of computer memory has become increasingly clear. Larger and larger programs are being written which require a structured data base to operate with any efficiency. Many of these could well benefit by replacing tedious searches with a fast, efficient, content-addressable access of the data store. A good example is the "key-word" library search. If one asks for a list of the books written by J. von Neumann, we do not expect the system to look at each title in its store and save only those written by von Neumann. And, if there happens to be a catalog prepared, designed to answer this particular question, we do not want to have to do a binary search to find the correct section of the catalog—we want to retrieve the answer directly!

There are many other problems which might find content-addressability advantageous. Examples abound in artificial intelligence where prohibitively large tree searches are encountered; question answering machines; graphics systems; and most conversational (time-shared) systems, which require immediate, direct access to a large data store to interact effectively. To date, most investigations into content-addressable memories have been concerned with hardware; such memories have not yet proved to be economically feasible. Even if they had, it is not clear that the obvious gain in speed would compensate for the loss in generality and flexibility. For the moment it can be said that software simulations are a stopgap measure. They are. But it is not certain that they will be completely replaced by hardware in even the relatively distant future.

It is also the function of a formal computer language to permit the problem programmer to phrase his algorithm in a natural manner that does not distract him. For many problems, it is most natural to talk about information as "relational triples;" e.g. in a graphics system one might want to say:

<Picture in> <Window A> is <Line B>;

or:

<Connected to> <Line B> is <Line C>.

The associative processor approach to content-addressability allows this.

The following example helps to clarify how an associative processor can be employed effectively to approximate content-addressability. Suppose we wish to know the phone number of the Acme Corp. It is a simple matter to look it up in the local phone book (performing a binary search!). It is, however, quite a different story to find out whose number is 763-6244 (using the same directory). An associative processor would find both tasks equal. In this example, the "association" is between a subscriber's name and his phone number. In translating this to a two-place relation, "phone number of" could be the relation, and using the <R,x,y> format we would say:

<Phone number of> <Acme Corp.> is <763-6244>.

This is a type of associativity wherein we may now directly reference this triple by any of its content-addressable components or combination thereof. If we use only the first component, phone number, as a key, what will be referenced is the entire phone book. If we specify the two components phone number and 763-6244, then we are referencing all associations containing the name(s) of the person(s) having the phone number 763-6244.

We are, of course, working with a conventional computer memory. The general strategy used to effect the simulation of an associative processor was hash-coding. For those unfamiliar with the term, hash-coding is simply a technique whereby an arithmetic transformation is applied to a "name" or designator to generate an internal address. Hash-coding (also called "scrambling," "scattering," "randomizing," etc.: for further information see [21,22]) by itself provides a restricted but significant approximation to content-addressability, but hashing alone does not provide any kind of associativity; and there is always the problem of the *collision*, i.e. when two distinct names hash to the same internal address: $X \neq Y$ and $H(X) = H(Y)$. Hashing partitions the space of names into equivalence classes. Hopefully each class has only one element (otherwise it is the *bucket* problem [22]), but two or more names may be equivalent under this partition.*

*Even restricting names to four letters of the English alphabet, a one-to-one transformation would require a table with 456,976 entries to guarantee no collisions.

By providing an interpretive language with an associative storage structure it is possible to achieve great flexibility. To this end we decided to use an existing interpreter and give it a new storage structure, rather than start at the bottom by designing a special purpose interpreter. Principally, we were concerned with the storage structure, and the vehicle for it was initially felt to be unimportant, since the structure relies on the host only superficially. In considering the question of the interpreter, we were faced with very little choice. A major consideration was that of availability; fortunately this consideration led us to the TRAC* language. It has proven to be a most elegant host, and credit for the power of the resulting system must be shared by both the interpreter and the structures (storage- and data-) given to it. However, we feel that the additional primitives are excellent vehicles which change the original processor into an efficient language for writing man-machine and machine-machine communication systems. Familiarity with the TRAC language will not be assumed in the sequel.

TRAMP is two packages of primitive functions that have been added to the TRAC language: one provides the inference mechanism and the other provides the document retrieval. These functional packages were machine-coded for embedding in the UMIST† interpreter

*TRAC is the trademark of Rockford Research Institute, Inc., Cambridge, Mass. in connection with their standard languages [15].

†UMIST [23] is closely patterned after the standard TRAC T-64 language and was implemented at the University of Michigan by Tad Pinkerton with the cooperation of Mr. C.N. Mooers, creator of TRAC T-64.

on the IBM 360/67 under the MTS executive [23]. Although this union has proved most fruitful, the associative processor is totally independent of the interpreter and actually relies on it only for I/O. In fact, a second version of the TRAMP associative processor is currently available, called MADAM [24], which can be used by any host using standard OS/360 calling sequences. The relational package is also independent of TRAC, except that it relies on the type of recursion that the interpreter provides. The relational package is totally dependent on the associative storage structure.

Feldman's initial work [18] was a strong motivation in the design of this system, and led us to adopt his notation, viz. the generic entity:

$$A \quad (O) \quad = \quad V$$

<$\underline{A}$ttribute> of <$\underline{O}$bject> equals <$\underline{V}$alue>.

Thus the *Associative Triple* is: <A,O,V>. Each of the three components is a non-empty set. To the associative processor this is an ordered triple but no interpretation or meaning is attached to the ordering* and all three are treated equally, giving none a priority. By appropriately designating the three components as being constant or variable, we can ask eight "questions"† of the processor. Again using Feldman's notation,

*This is in contrast to the relational package which places an artificial structure on the triple, viz. calling the first component a *relation* and the second and third its arguments.

†For the remainder of this chapter we shall be using the word "question" to mean a *document retrieval request*. This should not be confused with the "questions" asked of a question-answering program.

with a slight re-ordering, they are:

F0	A (O) = V
F1	A (O) = z
F2	A (y) = V
F3	A (y) = z
F4	x (O) = V
F5	x (O) = z
F6	x (y) = V
F7	x (y) = z

where {A,O,V} represent constants, and {x,y,z} are variables. Question F7 is not a question at all but a request for a dump of the associative memory, and in TRAMP such a dump is given. Question F0 simply asks: "Does A (O) = V ?" and the answer is a kind of truth value. In the case where A, O and V are all singletons, the truth value is a straightforward 1 or 0 denoting whether or not the specified association can be verified by the data. The interpretation is slightly ambiguous, however, when one or more of the three sets has cardinality greater than one. To illustrate, assuming that the associative sentence

COLOR (FLAG) = RED; WHITE; BLACK*

has been stored, these five questions have the following truth values:

*Since the comma already plays an important role as a TRAC language meta character, it is unavailable as a set element delimiter. Therefore the semi-colon (;) plays that role in TRAMP.

1.	COLOR (FLAG) = BLUE	0
2.	COLOR (FLAG) = RED;WHITE;BLACK	1
3.	COLOR (FLAG) = RED;WHITE;BLUE	?
4.	COLOR (FLAG) = BLACK	1
5.	COLOR (FLAG) = RED;WHITE;BLACK;BLUE	?

Questions 1 and 2 are clearly false and true respectively, but questions 3 and 5 are each partially true and partially false; question 4 is only half true. The interpretation which seemd most natural, and the one adopted by TRAMP, gives the truth values as shown, namely:

if ALL associations implied by the question are resident in memory, or derivable therefrom, the value is "1"

if NONE, the value is "0"

if SOME, but not all, the value returned is "?".

To say that these are the "values returned" in the context of TRAC is the same as saying that these are the three *return codes* of the function.

Questions F1-F6 simply ask the system to "fill in the blank(s)," i.e. to replace the variable with the set that is the answer to the question. For example, question F1 asks for the set of all V's that A (O) equals. Question F3 asks for the sets of all O's and V's that have a first component "A." Because of the recursive nature of TRAC, questions F1-F6 may be nested in any way, to any desired depth. One may ask: "How many fingers on a hand?"; "What figures are pointed to by the arrows in window Q?"; "How old are the fathers of the wives of Mary's brothers?"; or any questions composed in any way compatible with the stored data, nested to any level.

For those totally unfamiliar with TRAC, this section assumes only the syntax of a primitive function call. The sharp sign (#) signals the start of a function call, with the call itself enclosed in an immediately following pair of parentheses. The arguments are separated by commas, and the first argument is the name of the function. For example

```
#(sub,ARG)
```

is analagous to the Fortran

```
CALL    SUB(ARG)
```

3.1 ASSOCIATIVE STORAGE

The name of the storage function is *dr* and the syntax of the call is: #(dr,A,O,V). Again, the three arguments to *dr* are each non-empty sets. Each point in the cartesian product of the three sets is stored, i.e. each element of each set is grouped with each pair of elements of the other two sets, and the resulting triple is stored. Thus a single call on *dr* stores as many associations as the product of the cardinalities of the three sets. The storage sentence:

```
#(dr,AGE,JOHN;MARY,64)
```

would therefore store:

AGE (JOHN) = 64

AGE (MARY) = 64.

The actual storage is accomplished by pairing each A and O to point to a list of V's, each A and V point to a list of O's, and each O and V point to a list of A's. These "answer" lists are, strictly speaking, unordered, except that they retain the order in which they were stored. That is, asking the question:

"Whose age is 64?" or AGE (y) = 64

would now yield the answer:

JOHN; MARY *not* MARY; JOHN

It should be noted that this is a pure storage structure, and it does not deal with semantics; *dr* simply inserts associations into memory in a way that they can be quickly retrieved. TRAMP is not a question-answering system that checks for redundancies or inconsistencies of data.

3.2 DOCUMENT RETRIEVAL

The primary retrieval function has the name *rl*. The syntax of the function call is identical to that of *dr* except for the specification of variables. A *variable* in TRAMP is denoted by enclosing a name, possibly null, within asterisks (*). Thus #(rl,A,O,V) has no variables (F0) and asks whether A (O) = V; #(rl,A,O,*X*) asks: "What does A(O) equal?" If the variable is *named*, i.e. there is a name within the asterisks, then the function is *null-valued* (see Appendix) and the answer is stored in TRAC form storage labeled by the name: #(rl,A,O,*ANS*) would store the set of V's which A (O) equals under the label "ANS." If the variable is not named, e.g. #(rl,A,O,**), then the answer is the *value* of the function.

#(rl,COLOR,**,*COLOR*) is an example of a two-variable question with one named and one unnamed variable. The result in this case would be that the set of things having the attribute COLOR would be returned as the value of the function, while the set of colors is placed in form storage under the label "COLOR."

The two-variable questions (F3,F5 and F6) simply use the name table of one of the variables and index through that table, internally always asking the one-variable questions. An alternative would have been to supply a "use-list" (see [19]) so that two-variable questions could be handled as efficiently as one-variable questions. However, in practice it turned out that storage space was much more critical than CPU time so the choice was made to handle the two-variable questions in this "brute-force" method. Since the associative processor does not assign any priority to the three components, questions F3, F5 and F6, although relatively slower than the one-variable questions, are all equal among themselves. The process of answering a two-variable question in this implementation is less efficient because it must iterate on the one-variable questions, the number of iterations being a linear function of the size of memory.* The speed with which the one-variable questions are answered is not significantly affected by the size of memory! The three-variable question #(rl,**,**,**) is answered by an efficient enumeration of one of the association tables to give a full dump of the associative memory. Alternatively, one can call; #(dump).

*Here, and subsequently, "size of memory" refers to the amount of data in the structure, rather than the physical extent of the system.

Going back to the earlier example of the AGE's of JOHN and MARY, the question #(rl,AGE,JOHN;MARY,**) should have as its value 64;64. That is, redundancies can be valid and should be reported. But there are certain times, particularly in the two-variable questions, when redundancies become quite a nuisance (and even threaten to overflow the interpreter). Therefore, the function *rl* will always return an answer set with all redundancies deleted. A second entry point is provided, with the name *rlr*, which is identical except that it does not check for redundancies (making it considerably faster) but returns the answer "set" as it finds it.

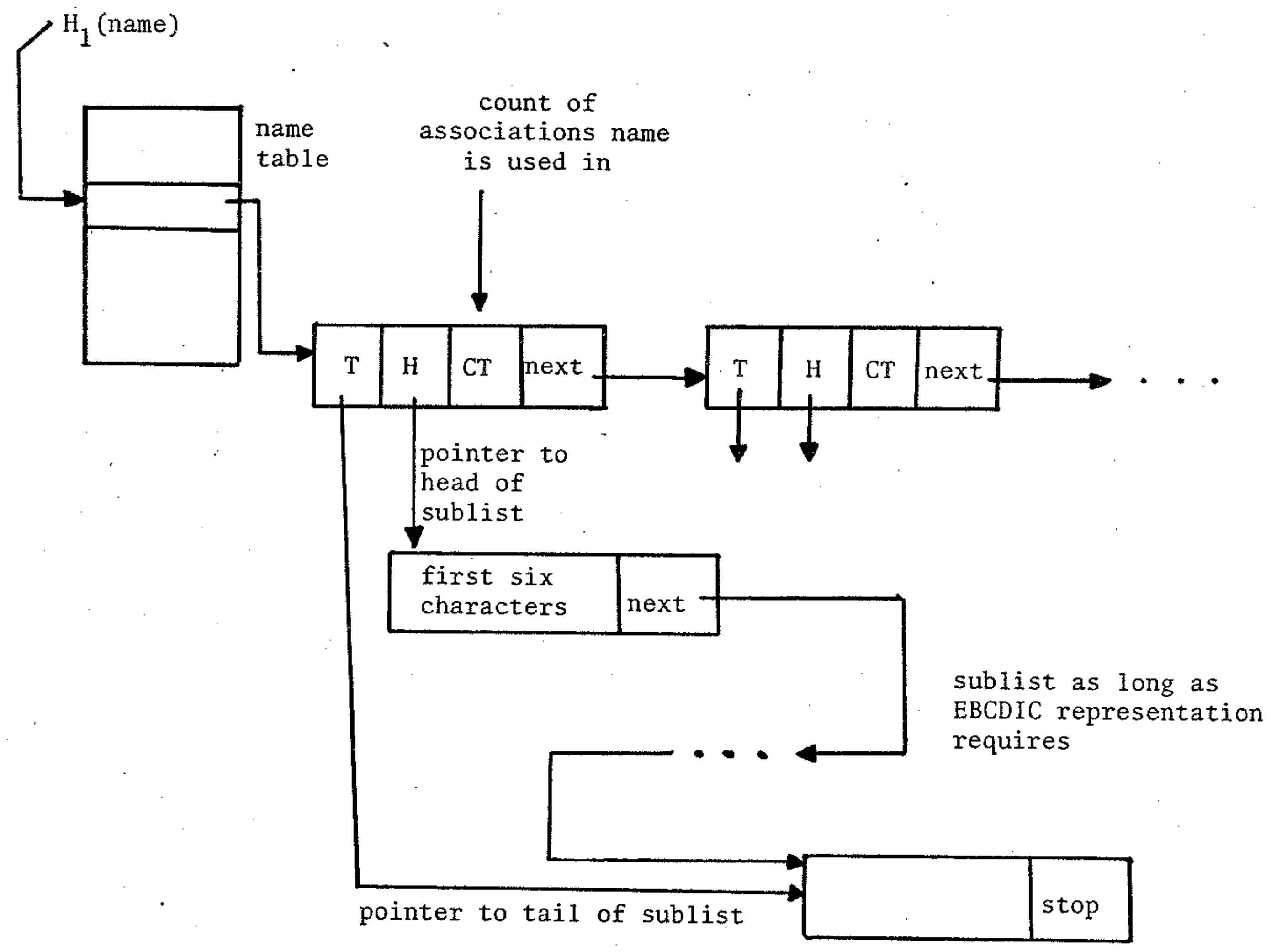
rl generates the union of the answer sets. That is, the question #(rl, AGE,JOHN;MARY,**) has two answer sets: the AGE of JOHN and the AGE of MARY. *rl* simply forms the *union* of however many sets there might be. *int* is the function (yet another entry point to the same routine) which generates the *intersection* of the several answer sets. Thus #(int,SOUTH;WEST,AUGUSTA,**) generates the set of all things that are *both* south *and* west of Augusta; #(rl,SOUTH;WEST,AUGUSTA,**) on the other hand, would generate the set of all things *either* south *or* west of Augusta.

3.3 IMPLEMENTATION STRATEGY

As stated earlier, hash-coding is the technique most basic to the associative processor design. We now discuss how this was implemented and how it operates. Before going on we would like to restate and clarify the relationship between TRAC and TRAMP (since there has always seemed to be confusion and uncertainty on this issue among those only casually acquainted with TRAMP). TRAMP is *embedded* in TRAC. That is, all of the TRAMP functions described in the Appendix were written in system 360 assembly language and they are now callable from within TRAC as if they were pre-defined primitives. In other words, TRAMP has extended TRAC by adding a number of functions and operations. The TRAMP user sees no operational distinctions whatever between the built-in functions provided by TRAC and the functions provided by TRAMP.

The associative processor uses three *name* tables and three *association* tables, one each for each of the three components of the associative triple. These tables are shown graphically in Figures 2 and 3 respectively. We shall explain these figures by stepping through the procedure that is followed for a typical storage operation. Suppose that the statement #(dr,WIFE,JOHN,MARY) is made. Each name that appears must be stored somewhere in memory. The full name must be present so that it can be retrieved and so that when it is referenced a collision can be identified and resolved. The first hash, H_1, then is applied to the "A" component ("WIFE") to generate a displacement from the *A name table*. The designated table entry is then inspected. If the entry is zero, then there is no collision and "WIFE" has never appeared before as an "A" component. Accordingly, the table entry is now made to point to the header for the name "WIFE." If the table entry is not zero, the header to which it points is inspected to see if it is the header for "WIFE." If so, the "A" name has been processed

Figure 2: NAME TABLES


H_1(name)
name table
count of associations name is used in
T
H
CT
next
T
H
CT
next
pointer to head of sublist
first six characters
next
sublist as long as EBCDIC representation requires
stop
pointer to tail of sublist

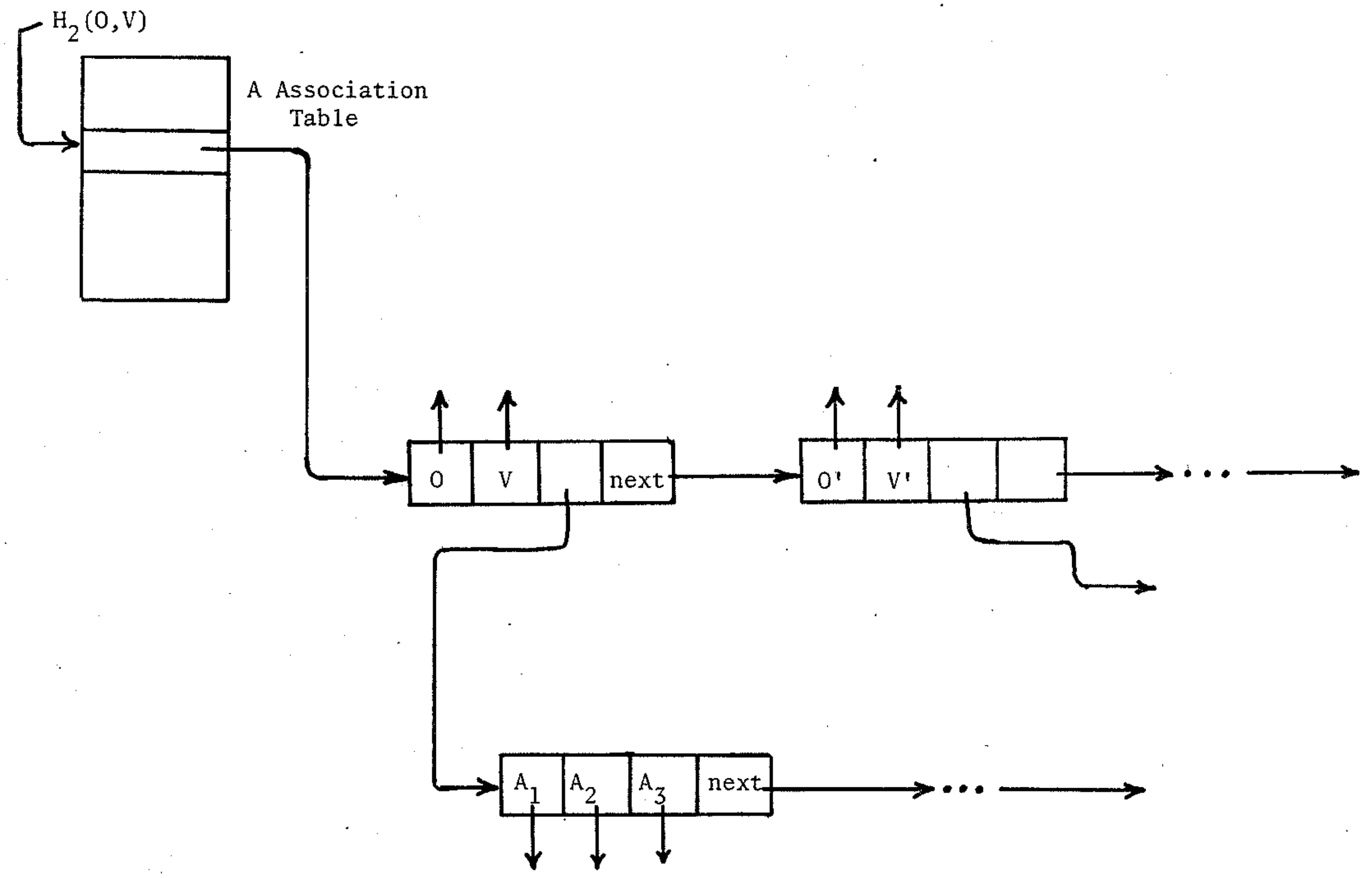
and we move on to the "O" component; otherwise there is a collision. For a collision, instead of a single header, there is an alphabetical[*] list of headers. Thus collisions are not really special cases: if there is no collision, then there is a list consisting of a single header, otherwise the list contains two or more headers in alphabetical order.

If the above process did not find the name, before it is actually placed in storage, a further check is made on the other name tables, thus avoiding redundant storage. Any name will appear at most once in memory, with up to four headers pointing to it.

The same procedure is applied to "JOHN" and "MARY," the "O" and "V" of this example, on their respective name tables. As a result of the name table processing, a unique pointer is associated with each of A, O and V, namely the pointer in the header which points to the location of the actual name. It is this unique pointer that will be used for the second hash, H_2. This second hash is a function of two arguments. Pairwise, in turn, the three components are now made to point to each other. It is this technique (with its inherent redundancy) that realizes the associative simulation. In our example, "WIFE" must now be placed on the *A association table*. To accomplish this the "O" and "V" pointers are hashed together to generate a displacement from the association table. To be able to identify collisions, both pointers that were used to generate the hash are stored in the header pointed to by the designated table entry.

[*] There is little significance to the fact that these lists are alphabetical. This was done simply because the "condition code" on the IBM 360 made it the natural thing to do. The advantages of it being ordered are negligible.

Figure 3: ASSOCIATION TABLE



Collisions are again resolved by ordered lists. The association table format (Fig. 3) has the table entry pointing to an ordered list of headers for O-V pairs. Each header then points to the associated "answer list," i.e. the set of all A's that have been associated with that O-V pair. The header contains the "O" and "V" pointers for identification, the pointer to the answer list, and a pointer to the next header on the list.

For the present example, "WIFE" is appended to the answer list by placing the unique pointer to "WIFE" at the end of the list. Note that H_1 is a function of the actual character string composing the name, while H_2 is only a function of where the name is stored and is independent of the name itself.

3.4 INTERNAL ORGANIZATION

In effect, TRAMP employs a triple storage technique to be able to reference an association in three different ways. Thus, the association A (O) = V is stored on each of the A, O and V association tables. This makes the answers to questions F1, F2 and F4 equally accessible and optimizes retrieval time.

TRAMP uses eight principal blocks of core. Though it is designed to run under a time-sharing supervisor which continually swaps TRAMP on and off a drum, TRAMP itself makes no explicit use of drums, discs, or other secondary storage devices; that is, such use by the system is transparent to TRAMP as well as to the user. Largely because of the executive under which TRAMP must operate no serious attempt was made to alter this situation. It would, of course, seem advantageous to employ some sophisticated paging scheme (such as [25]) but, happily, the present implementation seems not to suffer at all from having ignored the virtual memory problem.

The blocks of virtual core used are: four name tables [A, O, V and a name table for defined relations]; three association tables; and a General Storage list (called GS in TRAMP—analagous to "Available Space" in many list processors). GS provides all of the space for the associative memory, the space for relational definitions (Chap. 4), and all headers and lists. That is, all information indexed by the seven tables is found in GS. All scratch work and set accumulations (via unpacking the lists) is done in the "private control section" (system 360 "PSECT").

For purposes of illustration, let us follow the interpretation of:

#(dr,HUSBAND,EVE,ADAM)

First the name tables are processed. "HUSBAND" is hashed to produce a displacement from the "A" name table. The actual hashing scheme for H_1 is to form a full word (4 bytes) by concatenating the first, last, and middle two characters of the name, in that order. A single character may play only one of those roles, i.e. a name consisting of only one character has no last or middle characters. Any missing components are filled with zeroes. Thus HUSBAND forms "HDBA"; H_1(EVE) = "EEV"; and H_1(ADAM) = "AMDA." The full word so generated is then transformed, with the transformation being little more than squaring, mask and shift.

A list is generated in GS to hold the EBCDIC representation of the name: 6 characters (bytes) per double word and a 2-byte pointer to the next unit in the list. All units in GS are double-words (= 8 bytes = 64 bits). All lists in TRAMP are terminated with a zero or "stop" pointer. Each name list is terminated with a "stop" meta character. In the case of HUSBAND, two double-word units will be needed: the first will hold the 6 characters H-U-S-B-A-N, and a 2-byte pointer to the next unit which will hold the character "D", the stop meta character, and the stop pointer—wasting four bytes. Since HUSBAND is used here as an "attribute", we are concerned with the "A" name table. We look at the entry in this table designated by the hash. If this entry is empty, i.e. zero, there is no collision and HUSBAND has not been used previously as an attribute. If not empty, we look at the list of headers pointed to by the entry. Since the list is alphabetical, we need look only until we find HUSBAND or its alphabetical position in the list. Proceeding down this list of headers we compare the list generated above with the sublists pointed to be the headers. If a match is found, simply return this temporary list to GS and increment the "use" count for HUSBAND in its header. If it is not found, insert it ... after checking the O and V tables for its occurance. Previously HUSBAND may have been used as, say, an "object": #(dr,SEX,HUSBAND,MALE). In this case the HUSBAND name list would already be resident in GS. We therefore return the temporary copy of it generated above, and insert a pointer to the first list on the A table header list. Thus a name never appears in core more than once, though many pointers may point to it, including up to four headers if a name appears on all four name tables.

The above process is done for each HUSBAND, EVE and ADAM. The final pointer to the one name sublist of each is saved to generate the association table hash H_2. Let us follow the processing of the O association table. HUSBAND and ADAM ("A" and "V") are hashed together (multiplied, shifted and masked) to produce a displacement from the association table. The hash is performed on the two 2-byte pointers found during name table processing. The same sort of procedure is again followed to search the list of headers for a match, or insert a new header if necessary. This finds, or creates, the proper association list to which the pointer to EVE will now be appended (in general, of course, the list will not be a singleton, but even for this example TRAMP does not concern itself with polygamy). The association list—or answer list—elements consist of three 2-byte pointers to name sublists, the answers, and a 2-byte pointer to the next list element. This is a simple, unordered list, with new elements always being added to the right-hand end.

The entries of all the tables, as well as all list pointers within GS, point to double word units in GS. All pointers are two bytes long (16 bits), but are capable of addressing 128 pages of GS (1 page = 4096 bytes; 128 pages = 2^{19} bytes). For some applications this size is more than adequate, and for others not nearly enough. With its present scheme (addressing 2^{19} bytes with only 16 bits) TRAMP has an upper limit of 128 pages, which is a usable size for the majority of cases, including many AI applications. There is obviously a trade-off here since the more core that a pointer can address, the less percentage (though not proportionately less) of that core is available! There is a second trade-off because the size of the units which must be addressed determines the number of bits needed to address them—the larger the unit, the fewer bits required, but generally, the less efficiently it is used. We arbitrarily decided that the half-word pointers that TRAMP uses to address double-words are, in a sense, optimal. Should more experience prove us wrong, or if some special application should require much greater capacity, the structure could be augmented, e.g. to incorporate full 32-bit addresses, with little more than alteration of an assembly parameter. At this time it is not anticipated that explicit use will be made of any peripheral storage devices other than the transparent swapping performed by MTS. Without explicit utilization of peripheral storage, 32-bit addresses represent an upper bound.

The sizes of the various name and association tables are another assembly parameter. Currently the seven tables occupy four pages of core. This figure was arrived at arbitrarily also and will remain in force pending feedback which indicates that it is inappropriate.

TRAMP is initially loaded into core with all of its tables, a 1-page PSECT and eight pages of GS. Thereafter, when more space is needed TRAMP requests it of the system (MTS) in blocks of eight pages until the maximum of 128 is reached, or the system is unable to comply with the request.

TRAMP is continually generating temporary lists which are immediately returned to GS when no longer needed. As well, when an association is destroyed or a relational definition erased, the storage that is being released is at that time returned to GS making formal garbage collections unnecessary. However there is no obvious way of compacting the data—filling the holes—to minimize page faults. The only way to do that is to "dump" onto disc, erase the memory, and read it back in.

Chapter 4: THE RELATIONAL LANGUAGE

Thus far we have filled in two of the three components of the fact retriever depicted in Fig. 1: the "memory" box is realized by IBM 360 hardware and MTS software to afford a virtual memory using up to 32-bit addresses, of which we are currently utilizing 128 pages (19-bit addresses); the "document retriever" box is effected by the associative processor described in the last chapter.

We finally come to the component of most interest, the "logic" box used to infer implicit data from memory. In chapter 2 we saw the two principal approaches that have been taken: relational data structures and formal theorem provers. TRAMP takes the first approach, viz. it attempts to supply the appropriate data structure. The relational language to be described in this chapter was initially motivated by the need for data reduction and the observation of the natural compatibility of a relational data structure mapping onto an associative storage structure.

The software simulation of an associative machine, reported in the last chapter, provides a semblance of content-addressability and can be used to store and retrieve efficiently large amounts of data. However, any association, or combination of associations, will in general imply many more associations. For example:

[Husband (Mary) = John] ===> [Wife (John) = Mary]

$$\left.\begin{array}{l}\text{Father (Norm) = Harry}\\ \text{Brother (Harry) = Sam}\end{array}\right\} \text{===>} \left\{\begin{array}{l}\text{Uncle (Norm) = Sam}\\ \text{Nephew (Sam) = Norm}\\ \text{Brother (Sam) = Harry}\\ \text{Son (Harry) = Norm}\\ \text{etc.}\end{array}\right.$$

This situation can be resolved by requiring that the user redundantly enter his information in all the various ways that he might want to access it; or, more realistically, the user can define a *relation*, whereby he specifies what inference rules may be used in deriving the implied associations. Now the important distinction, once we have decided to admit relations, is whether their definitions will be *extensional* or *intensional*. We can extensionally define a relation by going through the structure and generating and *storing* all implied associations. For example this might be done with an iteration loop:

```
FOR HUSBAND (A) = B, LET WIFE (B) = A.
```

That is, all the ordered pairs which comprise the (binary) relation are generated and stored. This is the approach taken by many associative processors. The result is that all the implied information can in fact be made available, but there are two serious drawbacks:

1) In general, an extensional definition will gobble up extensive amounts of core (and peripheral storage), rendering it operationally uneconomical, or, in the extreme, infeasible.

2) Unless these iteration loops are entered frequently and regularly, their time dependency renders the extension incomplete and/or inaccurate.

The alternative to an extensional definition is the intensional definition where we characterize the relation, rather than exhaustively storing all of its ordered pairs. As an example, if we were to characterize the relation "Wife (of)" as:

WIFE = Converse of HUSBAND

and we now want to ask who is the WIFE of Harry, we could ask:

WIFE (HARRY) = ?

and the system, using the characterization given above, would expand the question to be:

WIFE (HARRY) = ? *or* HUSBAND (?) = HARRY.

It would ask both questions since it doesn't know if the desired association appears explicitly, implicitly, neither or both.

This is the operational strategy of the TRAMP intensional relational definitions. The user enters the definition as a sentence of a modified predicate calculus. This is the source program for a compiler whose output is an object program written in the associative language. This object program will then effect the "expansion" of questions to extract from the data those relevant associations that can be inferred from associations explicitly in (virtual) core and the intensional definitions of correspondences.

In operation, the user enters a *relational definition* as a sentence of the TRAMP relational language. The relational compiler is called to parse that sentence and output a program (the relational compiler is fully described in Chapter 6) . This defined relation is entered on the *relation name table*, with that table entry pointing to the compiled program which is held internally in GS. When a retrieval call is now made, via the function *rl*, a preprocessor, having found the name on the relation name table, will interpret the compiled program. This preprocessor expands the programs output by the compiler, filling in items specific to the call, like a macro-expander. This entire process is described in detail in chapter 5, but for now we shall only attempt to describe the strategy and tools. As an example, suppose that the following relational definition has been made:

$$\mathrm{SIS}(x,y) = \mathrm{FATHER}(x,z) \wedge \mathrm{FATHER}(y,z) \wedge \mathrm{FEM}(y) \wedge (x \neq y)$$

(which states that a sister of x is a y other than x who is female and has the same father). The compiler would output a program template to form the composition of FATHER with itself to generate a set of y's from a set of x's. That set of y's would then be intersected with a second set of y's generated by the unary relation FEM, and then the relative complement of the set of x's and the intersection of the two sets of y's would be formed. Suppose that now *rl* is called by:

```
#(rl,SIS,HARRIET,**)          [Who is Harriet's sister?]
```

This will, first of all, be handled exactly as previously described (as if the above definition had not been made) to retrieve any and all (explicit) associations that were made via

```
#(dr,SIS,HARRIET,hersisters)
```

Secondly, the program template would now be expanded. The expansion (described in Chapter 5) involves substituting HARRIET for "x", and ascertaining that the retrieval request is question type F1. The template must be manipulated by the preprocessor to answer the correct question of the eight possible (page 39). In this case the function call

```
#(rl,FATHER,HARRIET,**)
```

would be generated and the answer set to that question used in a second call for the relation FATHER:

```
#(rl,FATHER,**,#(rl,FATHER,HARRIET,**))
```

thus generating the first set of y's, namely the set of all things, including HARRIET, whose father is Harriet's father. Next the call is generated:

#(rl,FEM,**)[†]

whose answer set will be all those things that are female. These two sets are next intersected to form the set of all of Harriet's father's female offspring. Last, Harriet is removed from the set by the relative complement operation.

[†]Note that there are only two arguments to *rl* meaning that if the unary relation FEM has not been defined, the function call is syntactically invalid.

4.1 THE LANGUAGE DEFINITION

Relational *definitions* are made by calling on the function named *ddr*. The syntax of the call is:

#(ddr,(R = expression))

where R is the relation being "defined," and the "expression" is a sentence of a restricted predicate calculus, without explicit quantifiers, which "defines" the relation. The equal sign currently is read as implication to the left ("if expression, then R"), or, if you prefer, as set inclusion: the set of ordered pairs designated by the expression is taken to be a subset of the set of ordered pairs that comprise R. Notice that this is the sense in which the word "define" is used: the sentence defines a rule for determining set inclusion. The right side of the equation consists of relations, not necessarily distinct, with or without arguments, joined by the normal logical connectives:

TRAMP symbol	Logical symbol	Meaning
.A.	$\wedge$	conjunction
.V.	$\vee$	disjunction
.N.	$\sim$	negation

In addition there are two relational operators: *composition* (or relative product), denoted by a slash (/) and defined by:

$$(x)(y)[(R/Q)(x,y) \iff (\exists z)(R(x,z) \wedge Q(z,y))]$$

and *converse*, denoted by ".CON.", and defined by:

(x)(y)[R(x,y) <==> .CON. R(y,x)]

Finally, equality or inequality may be specified respectively by ".EQ." and ".NE.". The precedence of these operators is as shown below:

.CON.
/
.EQ., .NE.
.N.
.A.
.V.

The above precedence (descending order) ordering may be altered in the usual way by the appropriate use of parentheses.

The relational notation used by TRAMP is the <R(x,y)> format, where R is the relation and x and y its arguments. This can be read as: "y stands in relation R to x." The arguments are set off by parentheses. This is a slight distortion of the associative format, A (O) = V, but the ordering is preserved: R(x,y) corresponds to R (x) = y. With this as the source language, some typical definitions (primarily easily understood kinship relations) are shown in Figure 4. All definitions in Figure 4 show the argument to *ddr*, i.e. the sentence of the relational language, enclosed in parentheses, as the prototype above, This is due to one of the operational characteristics of the TRAC scanner. It is not always necessary, but never hurts and is simply good practice.

The first two examples of Figure 4 show the difference between the *abbreviated* and *expanded* formats. The problem of the compiler would be reduced to trivia if all definitions were abbreviated (that would be analogous to propositional calculus). The expanded format is necessary for

```
#(ddr,(BIGGER = BIGGER / BIGGER))  "BIGGER" is transitive

#(ddr,(BIGGER(a,b) = BIGGER(a,q) .A. BIGGER(q,b))
                                   exact same definition using the
                                   expanded format—specifying the
                                   dummy arguments.

#(ddr,(SIB = BRO .V. SIS .V. .CON.SIB))
                                   a sibling is a brother or a sister,
                                   and it is symmetric.

#(ddr,(BRO(cain,abel) = SIB(cain,abel) .A. SEX(abel,"male")))
                                   a brother is a male sibling.  Note
                                   that constants are enclosed within
                                   double quotes.

#(ddr,(MALE(x) = SEX(x,"male")))   defines the unary relation "MALE"

#(ddr,(BRO(x,y) = FATHER(x,z) .A. FATHER(y,z) .A. MALE(y) .A. x.NE.y))
                                   a brother is a male offspring of the
                                   same father, other than oneself.

#(ddr,(STEPMOTHER = FATHER / SPOUSE .A. .N.MOTHER))
                                   a stepmother is the spouse of the
                                   father who is not the mother.

#(ddr,(NEPHEW = SIBLING / SON))    a nephew is the composition of
                                   sibling and son.

#(ddr,(UNCLE = .CON.(SIBLING / SON)))
                                   in a male world, uncle is the converse
                                   of nephew and may be defined as the
                                   converse of the definition of nephew ...

#(ddr,(UNCLE = .CON. NEPHEW))      or simply as the converse of nephew.
```

FIGURE 4: SAMPLE RELATIONAL DEFINITIONS

several reasons: it is an important means of implicit existential quantification (explained below); the equality operators EQ and NE take as their operands the dummy variables of the expansion; and it adds an important facet to the language—easing the mathematical formality with which the programmer must view his relations. Many relations would be much more difficult, and some impossible (e.g. unary relations), to define without this feature, as exemplified by three of the four expanded definitions of Figure 4.

There are no explicit quantifiers. Quantification is handled in the following manner. On the left side of the defining "equation" are two dummy relational arguments, either explicitly named in the expanded format or else implicitly present in the abbreviated format. They are *free* variables. Any other dummy argument is *existentially* quantified. Existential quantification is used in two different ways. The first is composition, e.g.

GRANDFATHER = PARENT / FATHER

which in expanded form is

GRANDFATHER(x,y) = PARENT(x,q) .A. FATHER(q,y)

where the implicit quantification is:

$$(\exists q)[\mathrm{PARENT}(x,q) \wedge \mathrm{FATHER}(q,y) \implies \mathrm{GRANDFATHER}(x,y)]$$

q, the intermediary (link) is a *bound existential* variable. The "free" variables x and y may be thought of as *universally quantified*, since in a formal theorem proving framework the only wffs dealt with are closed wffs, and the closure S' of a wff S is formed by universally quantifying all free variables. S and S' are not equivalent, but they are intersatisfiable.

The second case is where e.g. the existential variable is used to require only that the free variable lies in the domain (or range) of some other relation. We illustrate this with the unary relation "AUTHOR" which we might define as: "AUTHOR(x) if x WROTE something" or formally as:

$$(\exists z)[\mathrm{WROTE}(z,x) \Longrightarrow \mathrm{AUTHOR}(x)]$$

This would be written in TRAMP as:

AUTHOR(x) = WROTE(z,x)

To repeat, the two (or one for unary) dummy arguments that appear on the left-hand side of the equation are free variables, and any other dummy argument is existentially quantified.

As an example now of how one might translate the dialogue of QA3 (shown on page 30), we suggest the following format. The reader should keep in mind that these translations which do not look much better than QA3's sentences, and perhaps a bit worse, will normally be TRAMP function calls that are generated internally by a TRAMP-coded program. The TRAC procedure definition (*ds* and *ss* described in Appendix A) are shown as they would appear in TRAC. They are slightly simplified in UMIST.

I) English version:

a) John is a boy; b) a boy is a person; c) a person is a human; d) a human has 2 arms; e) an arm has 1 hand; f) heredity property of "has."

II) QA3 version:

```
a) (IN JOHN BOY)
b) (FA(X)(IF(IN X BOY)(IN X PERSON)))
c) (FA(X)(IF(IN X PERSON)(IN X HUMAN)))
d) (FA(X)(IF(IN X HUMAN)(HAS X ARM 2)))
e) (FA(Y)(IF(IN Y ARM)(HAS Y HAND 1)))
f) (FA(X Y Z M N)(IF(AND(HAS X Y M)(FA(U)(IF(IN U Y)(HAS U Z N))
                                   ))(HAS X Z (TIMES M N)))).
```

III) TRAMP version:

```
    #(ds,SUBSET,(#(dr,SUBSET,X,Y)#(ddr,X=Y))) #(ss,SUBSET,X,Y)
a)  #(dr,ELEMENT,BOY,JOHN)
b)  #(cl,SUBSET,PERSON,BOY)
c)  #(cl,SUBSET,HUMAN,PERSON)
d)  #(dr,HAS,PERSON,ARM;ARM)
e)  #(dr,HAS,ARM,HAND)
    #(ddr,(HAS = HAS / HAS))
f)
    #(ddr,(HAS(x,y)=ELEMENT(z,x) .A. (HAS(z,y) .V. SUBSET(z,u).A.HAS(u,y)))
```

The two definitions given in (f) could also be expressed as:

```
#(ddr,(HAS = HAS/HAS .V. (.CON. ELEMENT)/(HAS .V. SUBSET/HAS)))
```

The counting function (*ct*, Appendix B) would then be used to answer the question: "How many hands does John have?"

```
#(ct,#(int,#(rlr,HAS,JOHN,**),HAND))
```

Examples of how these strange looking function calls can be generated internally are given in Chapter 7.

Chapter 5: SYSTEM OVERVIEW

We have now introduced all of the components of the TRAMP system. Perhaps the most interesting and non-obvious module of the system is the relational compiler which processes the language described in the last chapter. We shall defer discussion of that compiler to the next chapter, however. At this point it would be best to understand the relation of the various components to one another and see how they interact and function together to form a system. After this, the role of the compiler will be clear and we will be in a better position to discuss its operational characteristics and strategy of implementation.

Figure 5 depicts the lines of communication between the components of the system. We have omitted from Figure 5 the box which would be labeled "Auxiliary Functions" and would designate the 60 odd primitives of UMIST (including all of TRAC T-64) and the 20 odd primitives provided by TRAMP, which provide the environment for the system.

In Figure 5 we have the user of a TRAMP-coded system interacting with that program. This is a dynamic interaction since both TRAC and TRAMP were designed specifically for time-sharing, and in fact TRAC directly addresses itself to the "reactive typewriter" [15]. The TRAMP coded program then consists of a number of procedures (or *macros*,

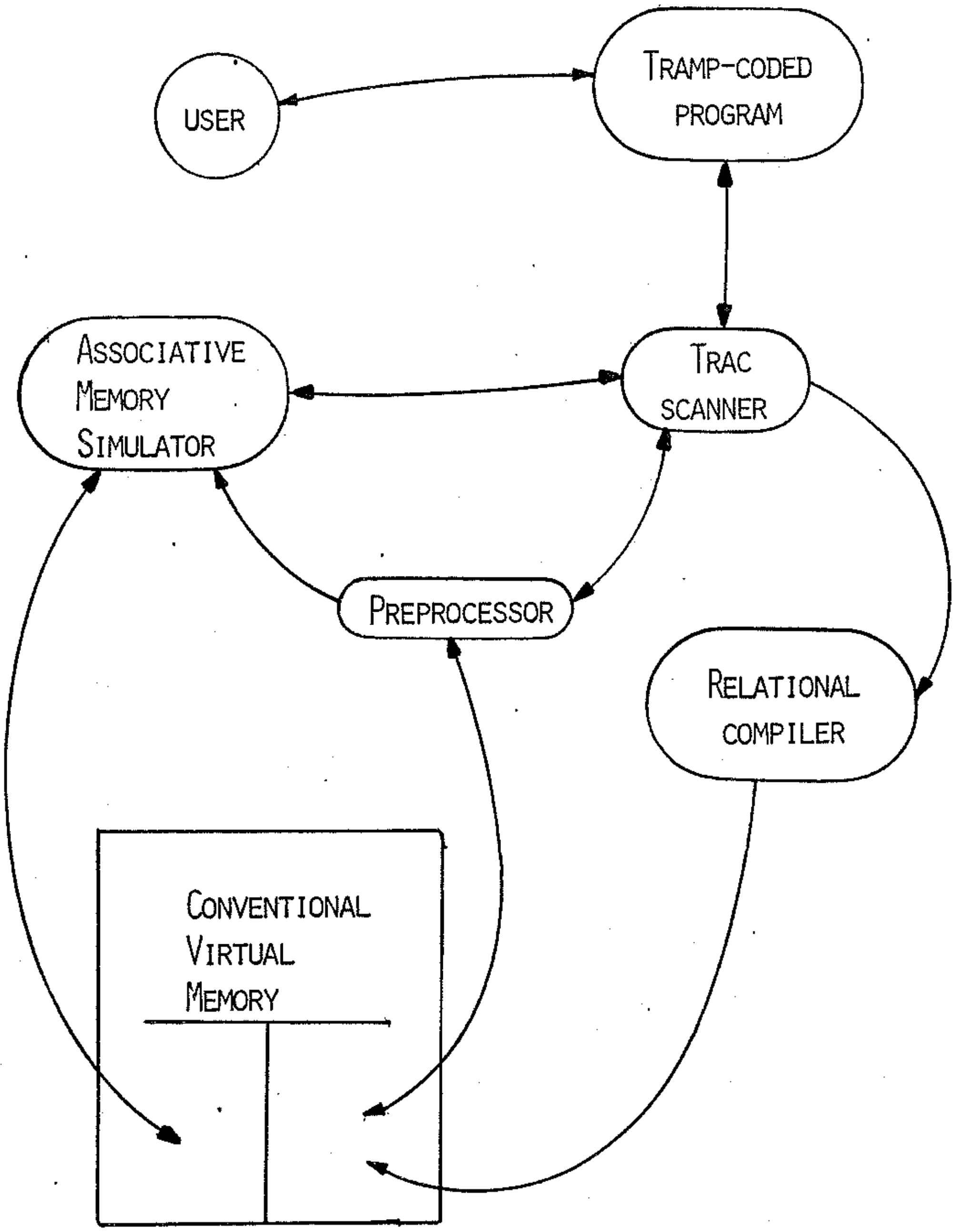


Figure 5: OPERATIONAL FLOW DIAGRAM OF TRAMP SYSTEM

since TRAC is technically a "macro-generator" language) which are actuated by the TRAC scanner. The scanner makes all "function calls" and receives and disposes of all "function values" and return codes. The scanner communicates with the Associative memory when it makes function calls to *dr* and *rle* (an entry to *rl*, see below). A function call to *rl* results in the scanner communicating with the preprocessor; and a function call to *ddr* invokes the relational compiler. The output of the compiler is a program template which is stored in a list oriented structure. The compiler has no *return* value and communicates with no other component. The box in Figure 5 labeled "Conventional Virtual Memory" is shown partitioned to indicate the logical division between that part of virtual core used for associative simulation and that part used for storing program templates. The preprocessor retrieves the program templates, interprets them (exactly like a macro-expansion) and then returns the expanded template, which is now a TRAMP procedure, directly to the TRAC scanner to be recursively scanned.

For most of the present chapter a working familiarity with TRAC would certainly be helpful. However it is not essential to being able to follow the TRAMP strategy for deduction. The crucial thing to know about TRAC is its manner of recursive evaluation and replacement. The TRAC scanner is alway scanning left to right the *active string*. When an *end of function* symbol is recognized (the right parenthesis that matches the *start of function* symbol, "#("), then that function is called. Every function returns a *value*, possibly null, which replaces the string of symbols constituting the function call on the active string. As an example, let the active string contain

```
THE SUM OF 3 AND 4 IS  #(ad,3,4).
↑
```

with the arrow denoting the current symbol being scanned. The scanner will keep moving to the right until the "#(" is encountered, signalling the start of a function (*ad* forms the sum of its two arguments). It notes this start of function and continues scanning. In this example nothing else will happen before the end of function is encountered, at which time it will call *ad*. The function value returned will be the string "7" and that value will replace the function call so that the active string will now contain:

```
THE SUM OF 3 AND 4 IS  7.
                        ↑
```

As no function is called before the end of function symbol for that function is encountered, if any of the arguments are functions, they will have been evaluated first.

Now let us return to TRAMP. The following simple example should help to clarify the system operation. Suppose that we have only the two facts:

```
JOHN IS HARRY'S FATHER
DORIS IS JOHN'S WIFE.
```

These two facts could be input in English, exactly as above, and parsed by the TRAMP coded program to produce the necessary function calls:

```
#(dr,FATHER,HARRY,JOHN)
#(dr,WIFE,JOHN,DORIS)
```

Chapter 7 gives examples of how such a translation can easily be effected. Now we wish to enter a few kinship relations which will give rules for expanding the data to include its implications. For example, we might want to express the relations of PARENThood and MOTHERhood. One way to do that would be:

```
    #(ddr,(PARENT = FATHER .V. MOTHER .V. PARENT/SPOUSE))
    #(ddr,(SPOUSE = HUSBAND .V. WIFE))
    #(ddr,(MOTHER = FATHER / SPOUSE))
```

[Note that we are using the convention that "R(x,y)" translates into "R of x equals y," and thus FATHER(HARRY,JOHN) means "the father of Harry is John." Under this convention the composition operator (/) may be read as "apostrophe s," e.g. a mother is the father*'s* spouse. One is quite free to adopt the convention that "R(x,y)" translates into "xRy" or "x stands in relation R to y" which would result in FATHER(HARRY,JOHN) meaning that "Harry is the father of John." If that convention were used, the order of composition would have to be reversed and the slash would be read as "of": MOTHER = SPOUSE/FATHER would translate to "a mother is the spouse *of* the father." A program may use either convention, or any other for that matter, but must of course be consistent.]

These definitions are presumably entered as "English-like" sentences and parsed to generate calls to *ddr*. The user may now ask questions such as:

```
            WHO IS HARRY'S MOTHER?
            WHO ARE HARRY'S PARENTS?
```

which would be passed to the scanner, respectively, as:

```
        #(r1,MOTHER,HARRY,**)    and    #(r1,PARENT,HARRY,**)
```

The scanner would then call the preprocessor. For the first question, involving the relation MOTHER, the preprocessor would retrieve the program template for MOTHER and output (back to the scanner) the string:

```
        #(r1,SPOUSE,#(r1,FATHER,HARRY,**),**)
```

(the string actually output is somewhat more involved than this, but this is an operational detail irrelevant at this point and defered to section 5.1). The scanner then processes this string and again calls the preprocessor for the relation SPOUSE. When that is expanded, the active string being scanned will be:

```
#(r1,HUSBAND,#(r1,FATHER,HARRY,**),**);#(r1,WIFE,#(r1,FATHER,HARRY,**),**)
```

This will be recursively evaluated by the scanner to effectively be:

```
        #(r1,HUSBAND,JOHN,**);#(r1,WIFE,JOHN,**)
```

since "JOHN" is the value of #(r1,FATHER,HARRY,**). That call will initially go to the preprocessor, but since there is no definition for FATHER, there will be no expansion and the call will be passed directly to the associative processor for direct retrieval of the association. Next, calls to the preprocessor are made for HUSBAND and WIFE, neither of which were defined and both result in direct associative retrievals. Since John of course does not have a HUSBAND, that associative retrieval results in a null set which will be joined with the set of his WIFEs, resulting in the "set of" Harry's mother(s), namely Doris.

The second question, "WHO ARE HARRY'S PARENTS?" will result in the same sort of expansions and yield the set {HARRY;DORIS}. To recap what has happened in this example, we have stored in the associative memory two facts relating John to his wife and son. We next entered some rules expressing correspondences among both primitive (FATHER, WIFE) and composite (MOTHER,SPOUSE,PARENT) relations. Then the question "WHO IS HARRY'S MOTHER?" was expanded to be: "WHO IS THE SPOUSE OF HARRY'S FATHER?" which was next expanded to: "WHO IS THE HUSBAND OR WIFE OF HARRY'S FATHER?" which yielded "WHO IS THE HUSBAND OR WIFE OF JOHN?" leading to the final answer: DORIS.

To recapitulate the interaction of the various modules of the TRAMP system as pertains to this example and illustrated in Figure 5, the process proceeds as follows. The user, presumably sitting at a terminal, enters his data in "English-like" sentences such as

JOHN IS HARRY'S FATHER.

and enters relational definitions, again in a pseudo-natural language,

A PARENT IS EITHER A FATHER OR A MOTHER OR THE SPOUSE OF A PARENT.

This is read by the TRAMP coded program and parsed to generate function calls to *dr* and *ddr* such as illustrated above. Those function calls are actually made by leaving the appropriate string, e.g. #(dr,FATHER,JOHN, HARRY), on the TRAC active string to be processed by the TRAC scanner. The scanner will then make the appropriate function call. In the case of *dr*, the scanner will call on the associative memory package; for *ddr* the scanner will call on the compiler; and for *rl*, the scanner will call the preprocessor. When the compiler is called, it will create a program template (or macro definition) and store it in a list structure. When the preprocessor is called, if the name of the relation does not have a definition (such as FATHER in our example), then the preprocessor just sends the call directly on to the Associative memory retrieval mechanism; if the name is found to have been defined, then the preprocessor retrieves the program template for it, expands that template, and returns the resulting TRAMP procedure to the TRAC scanner for recursive calls back to the preprocessor, with the recursion terminating on primitive (undefined) relations. The compiler does not allow circular definitions in the sense that A is defined in terms of B which is defined in terms of C ... which is defined in terms of A, so that this recursion will always terminate. (A relation defined in terms of itself, such as the above definition with PARENT = ... PARENT / SPOUSE, is recursive, not circular.)

5.1 DETAILS OF THE PROGRAM TEMPLATES

The above discussion of the operational behavior of the total system is sufficient for understanding intuitively how implicit information is effectively deduced. That is, we have explained how the output of the compiler will ultimately result in a TRAMP procedure used to infer data not explicitly present. However, there were certain inaccuracies in the explanation which we shall now rectify.for the interested reader. Others may skip this section and proceed to Chapter 6 without loss of continuity.

It is the operational strategy of the system that when a relation is defined, a rule has been given for expanding that relation. The relation may very well be a primitive as well. For example, we may start with the illustration above of John's father Harry, and Harry's wife Doris, and then define PARENT, SPOUSE and MOTHER. Later, likely in some other context, data may be entered that explicitly relates John and Doris: JOHN'S MOTHER IS DORIS. The point is that when the system is asked to retrieve a relation for which it has a definition, it does *not* presume anything about that relation except that it *might be* implicit. Specifically, the system takes the position that the desired association will appear either explicitly, implicitly, neither or both. Thus, when the preprocessor is invoked by:

```
#(rl,MOTHER,JOHN,**)
```

it will find the program template for MOTHER, as described above, but now it will also assume that the association may be stored explicitly. The system is always accumulating a *set* of answers and it tries to gather as complete a set as defined by the various definitions involved. In particular, it will retrieve everything it can directly and will use all of the information that it has for indirect retrievals (which led to the seeming absurdity of asking for Harry's husband).

However, the preprocessor having been called by #(rl,MOTHER,JOHN,**) cannot now return the same call to the scanner since that would result in an infinite recursion. For that reason the function *rle* is provided as being identical to *rl*, except that it bypasses the preprocessor proceeding directly to the associative mechanism. Thus, in the above example, the actual string returned to the scanner by the preprocessor would be:

```
#(rle,MOTHER,JOHN,**);#(rl,SPOUSE,#(rl,FATHER,HARRY,**),**).
```

The scanner will pass the first function call,*rle*, directly to the associative memory. The second call, to *rl*, will result in the expansion of the program template for SPOUSE.

The two function calls above are separated by a semi-colon, the TRAMP set element delimiter. This is because in TRAC the value of a function simply replaces the function call on the active string, so that after evaluation, we will have the concatenation of:

<mothersofjohn> <semi-colon> <spousesofharrysfather>

or in other words, a TRAMP set. This leads us to a second problem: what if there is more than one way of finding the same y such that R(x,y)? and what happens in this "union by concatenation" with the null set? To illustrate, if we have the necessary defintions for MOTHER and only the two original data items [FATHER(HARRY,JOHN) and WIFE(JOHN,DORIS)], then after fully evaluating the string we would be left with the answer set:

;DORIS

which has a needless semi-colon. On the other hand, if we now add the data item MOTHER(HARRY,DORIS), then the final result on the active string would be:

DORIS;DORIS

since the association occured both explicitly and implicitly. Naturally, for more complex relations, the situation can be much worse, with several empty sets, and/or many ways of inferring the same answer. For this reason, a special internal function is provided, which we shall designate by "@@", where @ is a non-graphic hexadecimal character. The function @@ "cleans-up" the result of the program execution by deleting duplicates created by multiple relational paths leading to the same R(x,y) pair. It also discards "dangling" semi-colons which will appear whenever one of the paths leads to a null set. Lastly, @@ will properly dispose of the final cleaned-up set as was designated by the variable construction (explained in Chapter 3 and described more fully in Appendix B).

There are also a few more internal functions (not described in the TRAMP user's manual, Appendix B) that are needed. For example, one is required for transitive relations. Clearly transitivity is a special case because of the mode of operation under which the expanded programs are executed. A transitive relation must call on itself recursively, but must not recurse forever! The function @*trn* (again, as throughout this paper, @ represents a non-graphic hexadecimal character) is used to form the transitive closure of a relation, calling on the relation recursively accumulating an answer set, and terminating when any single recursion adds nothing new to that answer set.

Symmetric relations are also slightly special cases which must call on themselves recursively exactly once. If we have the definition:

```
#(ddr,(RELATION(x,y) = RELATION(y,x))
```

then RELATION has been defined to be symmetric and the retrieval call

```
#(rl,RELATION,X,**)
```

must be expanded to:

```
#(@@,#(rle,RELATION,X,**);#(rl,RELATION,**,X))
```

that is, the stored association must be looked for in both "directions." But, of course, the above expansion would lead to an infinite recursion.

This is handled simply by having the compiler check for symmetry and so flagging the relation if applicable. Then when the preprocessor expands the relation, as above, a fourth argument is given to the function *rl* which serves as a signal to the preprocessor not to reverse the function call, though it will still effect any other expansion that the template designates. In particular, the actual expansion for RELATION, above, would be:

```
#(@@,#(rle,RELATION,X,**);#(rl,RELATION,**,X,@))
```

Through the use of non-graphic hexadecimal characters and full utilization of the TRAC scanner, all of the above has been implemented in such a way that not a single change was required to the UMIST program, and the TRAMP generated programs may call on procedures and generate names for forms and procedures, which will in no way interfere with the rest of the program coded in TRAMP by the programmer.

We will now give one final example, somewhat more complicated than the previous, and go through it in detail, showing exactly what would actually be returned to the scanner by the preprocessor. We will begin by initializing TRAMP with the following function calls:

```
#(ddr,(COUSIN = PARENT/SIBLING/OFFSPRING))
#(ddr,(PARENT = FATHER .V. MOTHER .V. PARENT/SPOUSE))
#(ddr,(SPOUSE = HUSBAND .V. WIFE .V. .CON.SPOUSE))
#(ddr,(OFFSPRING = .CON. PARENT))
#(ddr,(SIBLING = BROTHER .V. SISTER .V. .CON.SIBLING))
#(ddr,(SIBLING(x,y) = SIBLING(x,z) .A. SIBLING(y,z) .A. x.NE.y))
```

```
#(dr,FATHER,BOB,DAVE)
#(dr,WIFE,DAVE,ELAINE)
#(dr,BROTHER,DAVE,SAM;HAROLD)
#(dr,FATHER,SUE;JIM,HAROLD)
#(dr,COUSIN,BOB,CAROL;EMMY;SHARON)
```

We have thus put into our system a small family tree and a few kinship relations for filling in branches on that tree. We will now reproduce on the next page a "trace" of the active string resulting from asking for Bob's cousins. Each line will represent the contents of the active string just before a function call is made by the scanner. The arrow will indicate which character is being scanned. This example is quite tedious, but it illustrates just how the preprocessor expands a template by manipulating the template to answer the correct question (of the eight possible—only F1 and F2 are shown in this example), and properly gives *rl* its arguments: constants and variables. This example further displays the recursive replacement of TRAC and how it is used.

```
#(RL,COUSIN,BOB,**)
                  ↑

#(@@,#(RLE,COUSIN,BOB,**);#(RL,OFFSPRING,#(RL,SIBLING,#(RL,PARENT,
                        ↑
       BOB,**),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(RL,PARENT,BOB,
       **),**),**))
         ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(RL,SPOUSE,#(RLE,PARENT,BOB,**);#(RL,FATHER,BOB,**);#(RL,
       MOTHER,BOB,**),**)))),**),**))  ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(RL,SPOUSE,;#(RL,FATHER,BOB,**);#(RL,MOTHER,BOB,**),**))))
       ,**),**))                      ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(RL,SPOUSE,;DAVE;#(RL,MOTHER,BOB,**),**)))),**),**))
                                           ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(RL,SPOUSE,;DAVE;,**)))),**),**))
                            ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,#(RLE,SPOUSE,;DAVE;,**);#(RL,SPOUSE,**,;DAVE;,@);#(RL,
                                  ↑
       HUSBAND,;DAVE;,**);#(RL,WIFE,;DAVE;,**))))),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;#(RL,SPOUSE,**,;DAVE;,@);#(RL,HUSBAND,;DAVE;,**);#(
                                    ↑
       RL,WIFE,;DAVE;,**))))),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;#(@@,#(RLE,SPOUSE,**,;DAVE;);#(RL,HUSBAND,**,;DAVE;)
                                        ↑
       ;#(RL,WIFE,**,;DAVE;));#(RL,HUSBAND,;DAVE;,**);#(RL,WIFE,
       ;DAVE;,**))))),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;#(@@,;#(RL,HUSBAND,**,;DAVE;);#(RL,WIFE,**,;DAVE;));
                                         ↑
       #(RL,HUSBAND,;DAVE;,**);#(RL,WIFE,;DAVE;,**))))),**),**))
```

```
#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;#(@@,;;#(RL,WIFE,**,;DAVE;));#(RL,HUSBAND,;DAVE;,**)
                                        ↑
       ;#(RL,WIFE,;DAVE;,**))))),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;#(@@,;;); #(RL,HUSBAND,;DAVE;,**);#(RL,WIFE,;DAVE;,**)
                     ↑
       )))),**),**))

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;;#(RL,HUSBAND,;DAVE;,**);#(RL,WIFE,;DAVE;,**))))),**)
       ,**))                         ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;;;#(RL,WIFE,;DAVE;,**))))),**),**))
                                   ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       #(@@,;;;ELAINE)))),**),**))
                      ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,#(@TRN,(
       ELAINE;DAVE))),**),**))
                   ↑
```

[in this last line we have shown what will happen with the function @trn—the set of parents is ELAINE;DAVE.]

```
#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,#(@@,ELAINE;DAVE)
       ,**),**))                                                    ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(RL,SIBLING,ELAINE;DAVE,**),**))
                                                                  ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(@@,#(RCOM,#(@TRN,(#(RL,SIBLING,
       **,ELAINE;DAVE,@);#(RLE,BROTHER,ELAINE;DAVE,**);#(RL,SISTER,
       ELAINE;DAVE,**))),ELAINE;DAVE)),**))
                       ↑
```

[this time we shall not show expansions within @trn, since they do not actually happen that way anyway]

```
#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(@@,#(RCOM,SAM;HAROLD;ELAINE;DAVE,
       ELAINE;DAVE)),**))
                  ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,#(@@,SAM;HAROLD),**))
                                                      ↑

#(@@,CAROL;EMMY;SHARON;#(RL,OFFSPRING,SAM;HAROLD,**))
                                                    ↑
```

```
#(@@,CAROL;EMMY;SHARON;#(@@,#(RLE,OFFSPRING,SAM;HAROLD,**);#(RL,
       PARENT,**,SAM;HAROLD)))                                ↑

#(@@,CAROL;EMMY;SHARON;#(@@,;#(RL,PARENT,**,SAM;HAROLD)))
                                                        ↑

#(@@,CAROL;EMMY;SHARON;#(@@,;#(@@,#(@TRN,(#(RL,SPOUSE,#(RLE,PARENT,**
       ,SAM;HAROLD);#(RL,FATHER,**,SAM;HAROLD);#(RL,MOTHER,**,SAM;
       HAROLD),**))))))
                    ↑
[again, we do not show expansions within @trn]

#(@@,CAROL;EMMY;SHARON;#(@@,;#(@@,SUE;JIM)))
                                          ↑

#(@@,CAROL;EMMY;SHARON;#(@@,;SUE;JIM))
                                     ↑

#(@@,CAROL;EMMY;SHARON;SUE;JIM)
                               ↑

CAROL;EMMY;SHARON;SUE;JIM   ⊥
                            ↑
```

While this might appear to be an inordinate amount of work, it is accomplished very efficiently. All calls to the associative memory are extremely fast and the interpreter (preprocessor) is quite streamlined. The TRAC scanner does not scan the entire active string each time but always resumes where it left off. In fact, once something has been scanned, it actually leaves the active string and is placed on what is called the *neutral string*. While no statistics are available, the elapsed real time from input of: "WHO ARE BOB'S COUSINS?" to output of "CAROL;EMMY;SHARON;SUE;JIM" with MTS under normal load (20-25 users) would be under 5 seconds.

Chapter 6: THE RELATIONAL COMPILER

We have now seen the source language for the compiler in Chapter 4, the object language that the compiled code is to be written in (Chapter 3), and in the last chapter we saw how the object code will be manipulated so that the code, viewed as a TRAMP procedure, can be used by the preprocessor to "ask the right questions" of the associative memory, thereby inferring implicit data. We shall now describe how the relational sentences are parsed to produce the correct program templates.

Referring back to the discussion in Chapter 4 of how variables in relational sentences are quantified, there are two free variables appearing on the left side of the "equation," and any other variables that appear on the right-hand side are existentially quantified (meaning that they may be thought of as skolem functions of the free variables). Variable names in the language are, of course, not restricted, as they are treated as dummy arguments, but for the sake of standardization in the present discussion, we will make the convention that the free variables are named "x" and "y," and all the skolem functions of x and y will be denoted by other Latin letters. Further, we will use capital Latins to stand for relations, and all variables, both free and existentially bound, will be denoted by small Latins.

It is the operational strategy of the compiler that of the eight possible questions (page 39), the compiler will only be concerned with questions F1 and F2:

F1:	A (O) = ?	R(x,?)
F2:	A (?) = V	R(?,y)

Appropriate manipulation of these two questions by the preprocessor will allow the other six questions to be answered. The compiler, however, will only be concerned with the cases where the relation and one of its arguments is given. The programs written by the compiler will then be designed to retrieve the set of things which complete the triple. For example, given R and x, the program output by the compiler will find the set of y's such that R(x,y). From the discussion in the last chapter, it can be seen that since the output is for an associative object machine (which operates as described in Chapter 3), the object code of the compiler will have an "A" component (relation), and one of either the "O" or "V" components (x or y), and will associatively retrieve the other "O" or "V" component dependent on the question being type F1 or F2. For example, given the sentence:

$$R(x,y) = Q(x,a) \ .A.\ S(y,a)$$

for the question F1 the program will be given R and x. R will designate the program and x will be the initial *input* for that program, being input to relation Q. Inputting x to Q will "generate" a set of a's, namely the answer set to the associative question "Q (x) = ?". That set of a's will then be the input to the relation S to generate a set of y's, the final answer set of this program (the set of y's is the answer to the associative question: "S (?) = A", or here "S(?) = Q(x)").

Likewise, for question F2, the program would be given R and *y* and asked to find x: y would be the input to the program, being input to S and chaining through Q to generate the set of x's. For question F0 (no variables—a true or false question) the program would be given all three of R, x and y, and either F1 or F2 could be asked internally to generate an answer set. For example, if the question were

#(r1,R,X,Y)

then F1 could use R and X to generate a set of y's. The original true/false question would then be answered by intersecting the generated set of y's with the given set Y, i.e. determining whether or not the statement is true by seeing whether or not the set Y is contained in the set of y's. In the exact same way, question F2 could have been used to answer F0 by generating a set of x's. Similarly question F3 (where neither x nor y is given) could be answered with F1, e.g. by using the entire "O" name table as X, etc. (the domain or range of a relation may be computed from the "global" domain and ranges represented by the "O" and "V" name tables). Such manipulation as just described is performed automatically by the preprocessor.

Thus the task of the compiler is to determine what the *input* to a relation is and how the *output* (answer set retrieved from the relation and its input) is to be manipulated (e.g. intersected, composed, etc.). This is the primary task of the compiler. Significant to the translation process, the compiler must also determine certain semantic properties of a relation, as will be described in Sec. 6.3.

It should be obvious that for the purposes of determining which argument of the two is input and which is output, the pair of arguments may be considered an unordered pair, as their ordering gives no information whatever. If either argument is x or y, then there is no problem since x is always an input (for question F1—F2 is the exact dual and we shall not concern ourselves with it) and y is always the final output. Likewise in a compositional chain of arguments there is no problem as they must simply be chained together.

$$R(x,y) = A(x,a) \text{ .A. } B(b,a) \text{ .A. } C(y,b) \tag{6.1}$$

clearly dictates that x is input to A; a is input to B; and b is input to C. We can introduce complications into the above example by changing the defining equation slightly to be:

$$R(x,y) = A(x,a) \text{ .A. } B(b,a) \text{ .A. } C(y,b) \text{ .A. } D(a,c) \tag{6.2}$$

where we have added the relation D to our conjunction. In this case we would interpret the relation D as placing a constraint on the relation A. This is because we would consider the input to D to be c, the output would then be a and intersected with the set of a's generated by A and x. There is no ambiguity here since if a were the input to D instead of c, we would have a compositional chain (A/D) leading nowhere. Our context is that (for question F1) we are given x and look for a path to y: *all paths must end at y!* The above complication is not serious, but it does demonstrate that our language is not context-free. The two atoms "B(b,a)" and "D(a,c)" are syntactically identical, differing only in the arbitrary naming of "b" and "c." Further, since they appear in a conjunction, the "a" represents the same skolem function in each case.

Yet one relation takes "a" as input while the other outputs "a," depending on the context established by the other atoms.

We shall further complicate the issue by adding one more atom to our defining equation for R:

$$R(x,y) = A(x,a) \ .A.\ B(b,a) \ .A.\ C(y,b) \ .A.\ D(a,c) \ .A.\ E(c,y) \qquad (6.3)$$

This changes the context of relation D. On the one hand we can continue to interpret D as placing a restriction on the A relation, as above, and interpret the new relation E as placing a restriction on the final generated set of y's, i.e., the set of y's generated by (6.2) is to be intersected with the range of the relation E. Using the vertical bar to denote restriction, this interpretation is: $R = ((A|D)/B/C)|E$. On the other hand if we reverse the input and output of relation D from that of (6.2), then the chain A/D/E is formed and we have: $R = A/(B/C \wedge D/E)$.

Before leaving this example, let us add one more term to (6.3):

$$R(x,y) = A(x,a).A.B(b,a).A.C(y,b).A.D(a,c).A.E(c,y).A.F(x,c) \qquad (6.4)$$

We now have two complete chains (A/B/C and F/E) that lead from x to y, but there is the relation D joining the two chains together. It is clearly ambiguous which "direction" D is to go, i.e. which of its two arguments is to be the input and which the output.

A second instance of the context-sensitivity of the language arises from the fact that there are essentially two levels of operators: we have the "normal" operators, such as conjunction, which take as their operands relations; but now the relations themselves are a kind of operator, taking as operands the relational arguments. Ultimately it is these dummy arguments of the expansion that determine the context and hence the semantic interpretation of the higher level (logical) operators. For example, the meaning of the conjunction operator is dependent on the number of individual variables in common out of the four (two for each of its binary operands):

1) if 0 in common, then .A. is a no-op R(a,b) .A. Q(c,d)
2) if 1 in common, then .A. means composition: R(a,b) .A. Q(b,c)
3) if 2 in common, then .A. means intersection: R(a,b) .A. Q(a,b)

In Chapter 4 we saw that the language allows two formats: expanded and abbreviated, the difference being whether or not the relations are given arguments. For example relation R in (6.1) is defined in the expanded format. The exact same definition could have been entered in abbreviated form by deleting the arguments and using the relational operators:

R = A / .CON. B / .CON. C

(Note however that definitions (6.2) and (6.4) could not have been entered in abbreviated form.) The abbreviated form is provided as a convenience where applicable. Clearly it is not necessary since any abbreviated definition can be expanded, while the converse is not true. The abbreviated form is quite simple, being generated by a well understood [26,27] *operator precedence* grammar. The expanded form is not only not operator precedence, it is not even context-free. The significance of this can be seen by drawing an analogy to the logical calculi. The abbreviated form is strikingly similar to the propositional calculus, while the expanded form resembles exactly, save for quantifiers, a subset of the first-order predicate calculus. Certainly a "predicate" and a "relation" are just different ways of looking at the same thing. We define an n-place predicate as a mapping over a domain D such that $P: D^n \longrightarrow \{T,F\}$, while we define an n-place relation as being a subset of the nth cartesian product of D. They are clearly the same thing since given an n-place relation R it directly corresponds to the n-place predicate P according to the rule: $P(d_1,\ldots,d_n) = T$ if and only if $(d_1,\ldots,d_n) \in R$.

Indeed, the complexity of parsing the two formats is true to this analogy. The propositional calculus possesses several well known decision procedures and operator precedence languages are well understood; the predicate calculus can have no decision procedure and context-sensitive languages present problems in mechanical parsing that are barely understood at all.

Internally, the compiler will only work with expanded definitions since a first phase of the compiler will convert all relational sentences to a standard canonical form in which all relations have arguments. Thus the compiler cannot take advantage of the rather pleasant operator precedence properties of the abbreviated form. In fact, in the context of expanding an abbreviated sentence, the abbreviations are not even context-bounded. This is because of the composition operator. As an example, consider the three sentences and their expansions:

1) $R = (A\ .V.\ B) \longrightarrow R(x,y) = (A(x,y)\ .V.\ B(x,y))$

2) $R = (A\ .V.\ B)\ /\ C \longrightarrow R(x,y) = (A(x,a)\ .V.\ B(x,a))\ .A.\ C(a,y)$

3) $R = (A\ .V.\ B/D)\ /\ C \longrightarrow R(x,y)=(A(x,b)\ .V.B(x,a).A.D(a,b)).A.C(b,y)$

We see that all three sentences begin exactly the same "R=(A.V.B" but the relation A is given a different argument in each case. Further, note that the differences in these arguments is not just arbitrary naming of variables. In sentence (1) the output is the free variable y, while in (2) and (3) it is a skolem function of the free variables x and y! There is no way of determining the arguments, in general, until the end of the sentence is encountered, and thus it is not of bounded context [28].

Many standard languages, e.g. ALGOL, are not operator precedence or even context-free, but very simple changes will convert them to context-free languages, and even if minor changes do not result in an operator precedence grammar, it is sufficiently close that operator precedence techniques may be used. This is not the case here. For example consider the sentence:

$$R(x,y) = (A(x,a)\ .V.\ B(x,b))\ .A.\ (C(a,y)\ .V.\ D(b,y)) \tag{6.5}$$

This sentence is logically equivalent to:

$$R = A\ /\ C\ \ .V.\ \ B\ /\ D \tag{6.6}$$

It is not the canonical expansion of (6.6) but it is a valid expansion. Operator precedence techniques would attempt to work individually on each of the two parenthetical phrases in (6.5). This cannot be done, however, since out of context each of those phrases is meaningless. Furthermore, it is not immediately clear how the phrases should be distributed against each other since that results in terms such as "A(x,a).A.D(b,y)" whose meaning is not clear.

6.1 THE PARSING ALGORITHM

It has been illustrated above that the relational language that we must parse is at best pathological, and at least does not lend itself to the highly developed mechanical parsing techniques for formal languages [26]. We shall now describe our method of dealing with this problem, and in section 6.2 demonstrate that our algorithm is correct in the sense that it both terminates and terminates after having generated the program template that will realize exactly the intent (as described in Chapter 4) of the relational sentence source program.

The compiler is broken up into three logical phases. The first phase is necessary to the operation of the compiler within the system but does not perform any parsing per se. The first phase has the job of converting all sentences to the canonical form, performing almost all of the syntactic error checking, and accomplishing the linkages necessary for incorporating the program template into the total system, as described in the previous chapter.

The canonical form that the parsing mechanism will deal with is as follows. First, all abbreviated sentences will have been totally expanded. This means removal of the two relational operators *converse* and *composition* and the correct instantiation of dummy variable names for all relations. As indicated above, this task is not context bounded. Phase I must make a complete scan of the sentence to determine the context before it can rescan and perform the instantiations. The context so gathered is also used to effect the *converse* operator. For the simple case where we have:

R = .CON. S

it is obvious that .CON. simply inverts the order of the arguments of S (in this case from S(x,y) to S(y,x)). However, .CON. may be applied to parenthetical expressions of arbitrary complexity, in which case the entire expression must be viewed as a "sub-definition" of a single relation. Then the input and output of that "imaginary" relation must be identified and reversed, leaving everthing else unchanged. For example,

R = A / .CON.(B / C / D) / E

will first be expanded to

R(x,y) = A(x,a) .A. .CON.(B(a,b).A.C(b,c).A.D(c,d)) .A. E(d,y)

Now the composition B/C/D is considered as a sub-definition of an imáginary relation which takes in this case "a" as its input and outputs "d". Thus a and d are switched yielding:

R(x,y) = A(x,a) .A. B(d,b) .A. C(b,c) .A. D(c,a) .A. E(d,y)

For this operation of removing the *converse* operator, we again require context to determine the input and output in order to know which variables to reverse. (Of course, the alphabetical ordering above and in most of the examples of this chapter, does not give any information as it might seem to do in these examples.) One cannot, as might seem at first to be the case, apply the .CON. operator by simply pairwise inverting all arguments in the expression, rather the expression must be viewed as a sub-relation.

In addition to removing the two relational operators, the canonical form will convert the equality operators to normal looking relations. The input format for the equality operators is: "x.NE.y". This will be converted to "@NE(x,y)" which is an internal pseudo-relation. The non-graphic "@" serves as a flag that this is a pseudo-relation, not to be translated into a retrieval call, and of course precludes interference with user defined relation names.

Our canonical form also requires that the sentence be in *disjunctive normal form* (*dnf*, an OR of ANDs—the exact dual of conjunctive normal form described in Chapter 2). This form will eliminate the apparent ambiguity of the scope of an existentially quantified variable (or in more familiar terms, the scope over which a dummy variable is *local*). Within any one disjunct any one variable name will always refer to the same thing; between disjuncts the same name will refer to different things. Thus in the sentence:

R(x,y) = A(x,a) .A. B(x,y) .V. C(x,a) .A. D(a,y)

we have the "a" as argument to relations C and D being the same, but having no connection with the "a" in relation A.

Disjunctive normal form is also extremely useful because of the environment that the program is to operate in. Disjunction is accomplished, as was seen in Chapter 5, simply by concatenation; conjunction is accomplished in the sense of intersection by the *int* function

(Appendix B), and in the sense of composition by recursive nesting. In dnf, the compiler can consider each *clause* (a clause is a disjunct—the exact dual of the clauses discussed in connection with resolution, Chapter 2) independently, out of context of the other clauses, with the several clauses being joined (ORed) together by simple concatenation.*

The last function that Phase I performs is the list processing necessary for incorporation into the TRAMP system. The program templates, once generated, will be stored in normal TRAMP lists (in GS) and they must be accessible, requiring that conventions be respected. The list processing that is performed is illustrated in Figure 6. In that figure the information stored includes two seperate programs P1 and P2. The compiler, it was stated, considers the cases where the relation and one argument is given. There are two such cases and each generates a program. We shall restrict our discussion entirely to program P1, which is designed to answer question F1, since P2 is just its dual.

Phase II of the compiler performs the actual parse, which, as pointed out above, is largely a question of deciding which argument is the "input" of the associative retrieval call, and which argument represents the generated answer set of that call. After this has been

*The compiler is presently being run without the section of code that performs the conversion to dnf. This is due to the bulk of that code. There is no theoretical problem whatever in algorithmically performing this conversion. This is not much of a hinderance in practice, since almost all user definitions are already in dnf, unlike the pathological examples of this chapter. Note that in Fig. 4 all of the examples are in dnf.

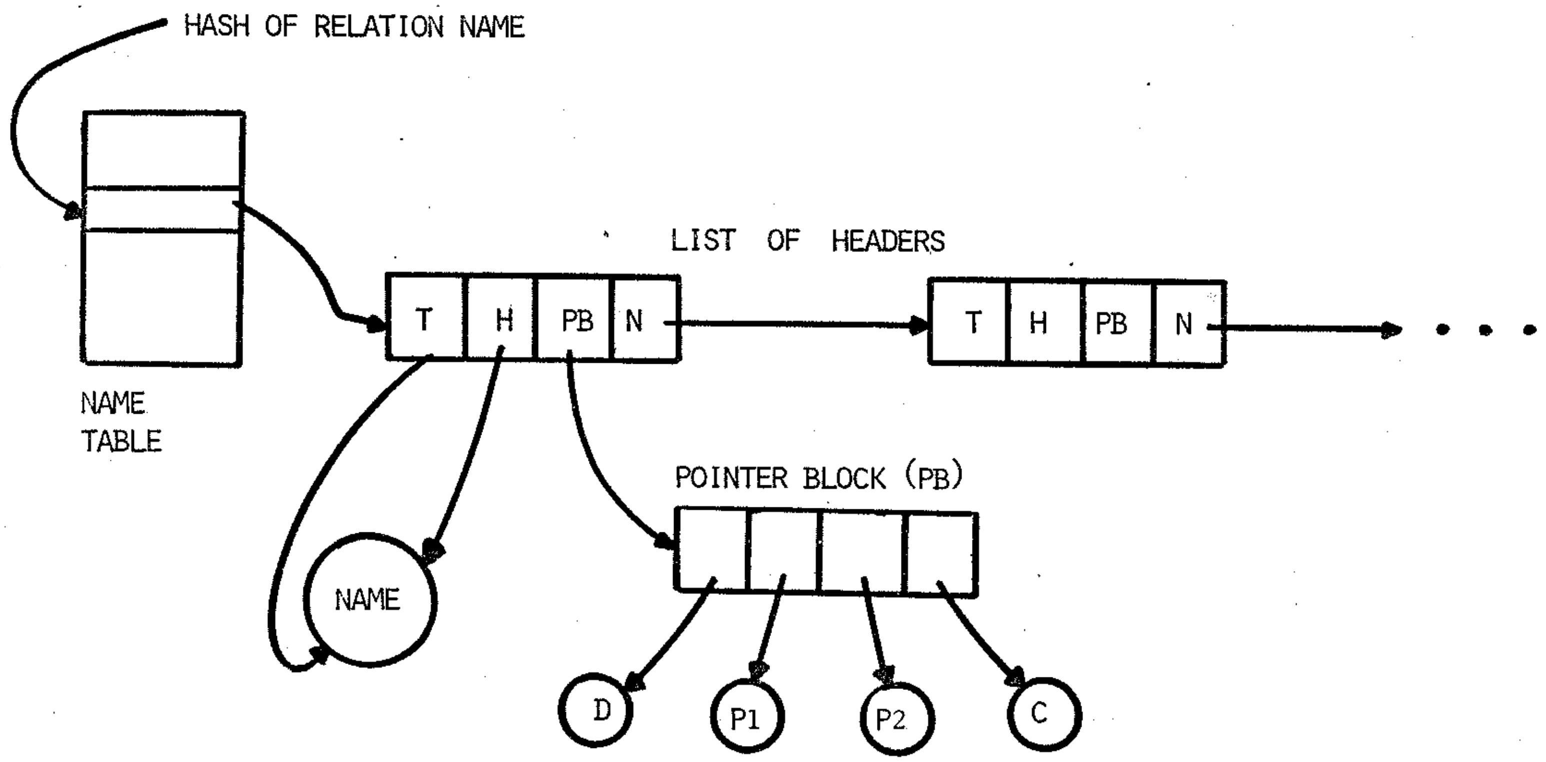


Figure 6: NAME TABLE FOR RELATIONS

The name of the relation is hashed to designate an entry in the defined relation name table. That entry points to a list of headers. Each header has four pointers: H and T are the pointers to the Head and Tail respectively of the sublist holding the name; PB points to the Pointer Block, described below; and N is the pointer to the next element on the list.

The Pointer Block also contains four pointers: the first, D, is the pointer to the EBCDIC text of the definition as entered by the user (for editing and displaying); P1 and P2 are the pointers to the two programs output by the compiler; C is the list of relations used to define this relation, permitting a check for circularity.

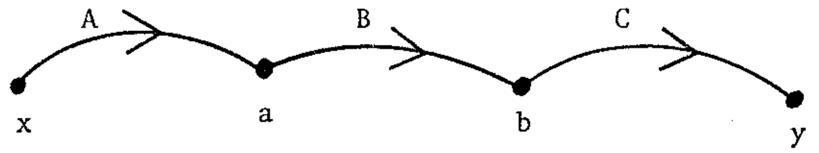
decided, phase III performs the code generation and analyzes the sentence semantically to determine various properties of the relation in question, e.g. transitivity.

The introduction to the present chapter pointed out some of the problems with mechanical parsing of the relational language, and the lack of any developed and applicable linguistic theory to fall back on. We shall rephrase our description of the parsing that is to be performed, and then we shall see that the chosen description is so powerful that it actually embodies the solution of how the parse can be accomplished!

6.1.1 REFORMULATION OF THE PROBLEM

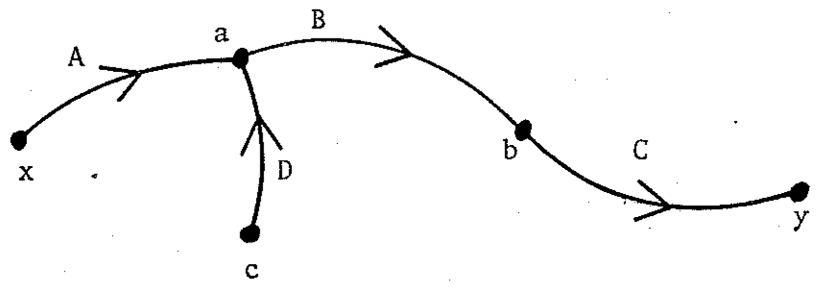
We shall consider the relational sentence to be represented by a directed network. Each individual variable, x, y and the skolem functions of x and y, will be represented as a node of the network. Each relation will be represented as a directed line between the two nodes representing its arguments. The source of the network will be the node labeled x, and the sink of the network will be the node labeled y. In this framework it can be seen that the associative retrieval calls that must be generated as the object code, are directly represented by the directed lines. R (x) = ? is the associative question that takes R and a set x and generates an answer set; this "generation" is just the directed line, with the node from which the line leads being the input and the node to which it points being the generated set. The intent of the object program is to take a set X and a relation R and generate a set of Y's. This will simply be the set that comes from starting at the source of the network, x, and tracing all paths to the sink, y. For example, the sentence (6.1) has the graphical representation:

[R(x,y) = A(x,a) .A. B(b,a) .A. C(y,b) (6.1)]



The sentence (6.2) is graphed below, with c being a "spurious" source, since no line comes into c.

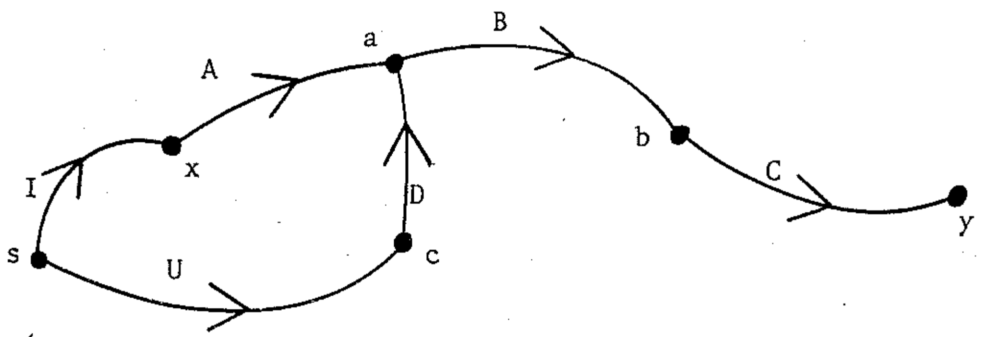
[R(x,y) = A(x,a).A.B(b,a).A.C(y,b).A.D(a,c) (6.2)]



This is clearly an accurate formulation of the parsing problem, but it also solves the problem! Just construct the graph and, knowing that all sentences are in dnf (so that only clauses ever need be considered, meaning that the only operator is the conjunction operator), we may interpret the resulting network as follows. The node y will be the only node in the network that is a sink, it is the final answer. For all other nodes, if only one line is incident to the node then it is a source (there may also be other nodes distinct from x with more than one line incident to it that is a source—this condition is clarified below). The node x is always a source. In the usual way [29], we shall do away with multiple sources by creating a single, imaginary source that feeds all other "sources." In our context we shall do this by means of two new relations, I and U, defined by:

I (x) = x	*identity relation*
U (x) = everything	*universal relation*

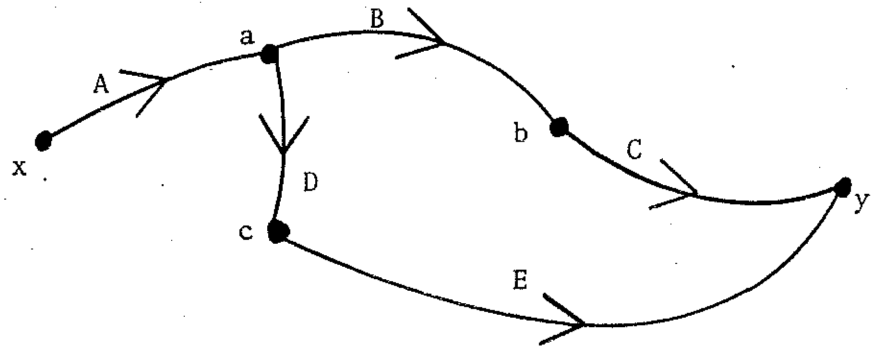
The single imaginary source will be labeled *s*, and there will be the line I(s,x) and for any other source p, there will be the line U(s,p). Thus we would redraw the graph for (6.2) as:



Henceforth we shall not distinguish between s and x, but simply refer to *the* source, assuming that s has been created if necessary.

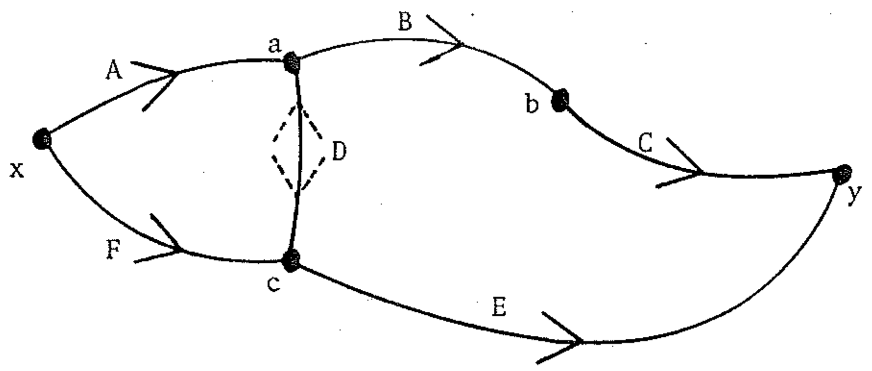
Given this network formulation of the sentence, we interpret the network by examining the configuration of the arrows at each node. Every node, except the imaginary node s, will have at least one line directed into it, and every node except y will have at least one line directed out from it. Our interpretation is that we "collapse" the sets of lines directed in and out and consider the single collapsed line going out as meaning the composition with the collapsed line coming in. The several lines coming in are collapsed by simple set intersection; the several lines going out are collapsed by "domain intersection," accomplished by sending the set elements along the several paths one at a time, taking the set intersections of the paths where they meet (they must meet, at least at y), and forming the union of these intersections over all elements of the incoming set.

Phase III will recursively trace out paths of the final network to produce the required object code. The task for phase II is now reduced to simply directing the lines of the network properly. This is truly a significant reduction of the problem, but are we really any better off? The problem of correctly directing the lines does not appear to be all that simple, in general. For example, sentences (6.2) and (6.3) apparently demonstrated a situation where the added context of a single atom might change the direction of one of the lines:



(6.3)

Sentence (6.4) demonstrated a true ambiguity in that there is no apparent way to decide which way the line for relation D is to be directed:



(6.4)

We shall prove in section 6.2 that these apparent difficulties in fact vanish, and the direction of the lines is easily and correctly computed by the algorithm presented in section 6.1.3. The algorithm as stated tacitly assumes that the original sentence is in dnf and is only applied to a single clause, which may be taken out of context. The algorithm further assumes the remaining conventions regarding canonical form. The pseudo-relations (@NE) are ignored during construction of the network and are easily handled by phase III.

6.1.2 SOME NEEDED DEFINITIONS

For the purposes of defining the algorithm, as well as more precisely clarifying some of the graph theory terminology used in the preceeding discussion, we shall now introduce some necessary formal definitions. A few of these definitions are peculiar to our purpose, but mostly they are semi-standard (paralleling a standard reference such as [30]) and presented here for precision of terminology.

DEFINITION 6.1: A *graph* is an ordered pair <S,R>, where S is a set of elements called nodes (or points, vertices, etc.) and R is a symmetric, irreflexive, binary relation over S. The graph is drawn such that a point represents each element of S, and for each unordered pair (s_1,s_2) for $s_i \in S$, a line (also called an arc, edge, etc.) is drawn between s_1 and s_2 just in case $(s_1,s_2) \in R$.

DEFINITION 6.2: Two nodes joined by a line are said to be *adjacent* to one another and each is said to be *incident with* the connecting line. The line joining the two nodes p and q is often denoted (p,q).

DEFINITION 6.3: A *directed graph* (or *digraph)* is a graph in which the relation R is asymmetric. The digraph is drawn the same as a graph, except that each pair of points is an ordered pair, and the ordering is denoted by assigning a direction to each line. The direction is specified by drawing an arrow, and the line (p,q) is *directed from* p *to* q, and said to be *incident (out) from* p and *incident (in) to* q.

DEFINITION 6.4: A *multi-graph* is a graph in which more than one line may be drawn between the same two points, often called a *graph*.

DEFINITION 6.5: The *degree* of a node in a graph is the number of lines incident with that node. For a digraph, the *indegree* of a node p, written *ID(p)*, is the number of lines directed into p; the *outdegree* of p, written *OD(p)*, is the number of lines directed out from p.

DEFINITION 6.6: A *network* is a digraph with two sets of distinguished nodes called *sources* and *sinks*. A source has positive degree and zero indegree. A sink has positive degree and zero outdegree. All other nodes are called *intermediate* nodes.

DEFINITION 6.7: A *path* is a sequence of alternating nodes and lines, $<N_1,L_1,N_2,L_2,\ldots,L_{n-1},N_n>$, beginning and ending with a node, such that for each i, $1 \le i < n$, the line L_i is drawn between the nodes N_i and N_{i+1}. A path is normally written simply as as sequence of adjacent nodes, the lines being understood. A *directed path* in a digraph is a path in which each line L_i is directed from N_i to N_{i+1}.

DEFINITION 6.8: A *cycle* is a path with at least two distinct nodes, that begins and ends at the same node. In a digraph, a *directed cycle (*or *cycle)* is a directed path that begins and ends at the same node; an *undirected cycle (*or *semi-cycle)* is an undirected path that begins and ends at the same node. A graph that contains no cycles is said to be *acyclic*.

DEFINITION 6.9: A graph is said to be *connected* if there exists a path between every pair of points. A digraph is said to be *strongly connected* if there exists a directed path between every pair of points; it is *weakly connected* if there is an undirected path between every pair of points. A graph that is not connected is *disconnected*. A network is said to be *N-connected* if every intermediate node lies on a directed path from a source to a sink.

DEFINITION 6.10: A *cut-point* is a node in a (weakly) connected graph, which if removed would disconnect the graph.

DEFINITION 6.11: An *isolated cycle* is a cycle, one of whose nodes is a cut-point, and removal of that cut-point would disconnect the graph into two *components* (or more), one of which would consist only of the remaining nodes of the cycle.

DEFINITION 6.12: A metric is placed on a graph by defining the *distance function D(p,q)* for each pair of points p and q, as being the length (number of lines) in the shortest path between p and q. For a digraph, D(p,q) is the length of the shortest directed path. For all p we have D(p,p) = 0; for all adjacent nodes (p,q) we have D(p,q) = 1; and for p and q not connected we have D(p,q) = ∞. For a network we define for all nodes p, *D(p)* is the minimum over all y in the set of sinks, of D(p,y).

DEFINITION 6.13: A *bridge* of a digraph is a directed path B whose end points u and v respectively lie on directed paths U and V, such that U and V have exactly two nodes in common, p and r, and such that u, v, p and r are four distinct nodes. (The two paths U and V thus form a semi-cycle, with B "bridging" across that semi-cycle.)

6.1.3 THE ALGORITHM

We start with an undirected graph G which represents a single clause of the canonical sentence. Each individual variable appearing in the sentence is a node, and each relation is a labeled line between the two nodes representing its arguments. We shall transform the graph G, first into a simpler graph G', then into a directed graph G'', and then into a final network G''' which will be the input to phase III. (In what follows, "graph" means "multi-graph.")

STEP I)

The graph G is converted to a smaller graph G'. All intermediate nodes with degree 2 represent composition. Any intermediate node with degree 2 may be deleted along with its two incident lines, and its two adjacent nodes joined by a new line. This compression may well result in a multi-graph. Any compositional chain that collapses to a point in this process clearly represents an isolated cycle and is noted for handling in step (IV). Any intermediate node of G' (after all other reductions have been made) that has degree 1, is deleted for handling in step (V). The resulting multi-graph G' is undirected and has no intermediate node with degree less than 3.

STEP II)

We shall now construct a sequence of sinks Y_i as follows. We initialize $Y_0 = \{y\}$, the single sink of the original network. Iteratively we compute the sequence $\langle Y_i \rangle$ by the rule:

$$Y_{i+1} = \{p \in (G' - \bigcup_0^i Y_j) \mid (p,q) \in G' \wedge q \in Y_i\}$$

As each Y_j is computed by this rule, each line (p,q) for $p \in Y_j$; $q \in Y_{j-1}$, is directed from p to q. Clearly, for some k, $Y_k = \phi$, at which time if $\bigcup_0^k Y_j \neq N(G')$ (the set of nodes of G'), then G' is disconnected and it is rejected. The resulting digraph is G''.

STEP IIa)

Step (II) does not specify how the line (p,q) is to be directed in the event that $p \in Y_i$ and also $q \in Y_i$ for some i. This condition often identifies a bridge and is ambiguous. We shall resolve the ambiguity by placing an ordering on the nodes of G'. Let $Q^i = \{q \in Y_i\}$; let Q^i_j be the jth component of Q^i (i.e. the jth equivalence class of Q^i induced by the equivalence relation: $p \equiv q$ if and only if there exists a path from p to q in the sub-graph Q^i). Within each Q^i_j assign each node q the number A(q) which is its alphabetical order within Q^i. For each $q \in Q^i$ we can estimate* its indegree by subtracting its outdegree from its degree. Clearly, for $q \in Q^i$ we have $D(q) = D(q,y) = i$. We now define $f(q) = (D(q), ID(q), A(q))$, and the ordering relation $\succ$ by: $q \succ p$ if and only if f(q) is lexicographically greater than f(p). Now, for $p,q \in Q^i_j$, we direct the line (p,q) from p to q if and only if $p \succ q$.

*This can only be an estimate because of course we do not yet know how the other lines will be directed. However this is quite sufficient and is only used because often it will produce better code.

STEP III)

The result of steps I-IIa is a directed, but also compacted, version of the original graph G. We now expand it to include all of the nodes and lines of the original graph. Nodes with degree 1 are replaced with the single line directed out from the node. Isolated cycles are handled in step (IV). The only other nodes that were deleted were those with degree 2. Such nodes were replaced by new lines which are now directed. We simply switch back: the node and its two incident lines replace the artificial line; the direction of the composition is the same as the direction of the artificial line.

STEP IV)

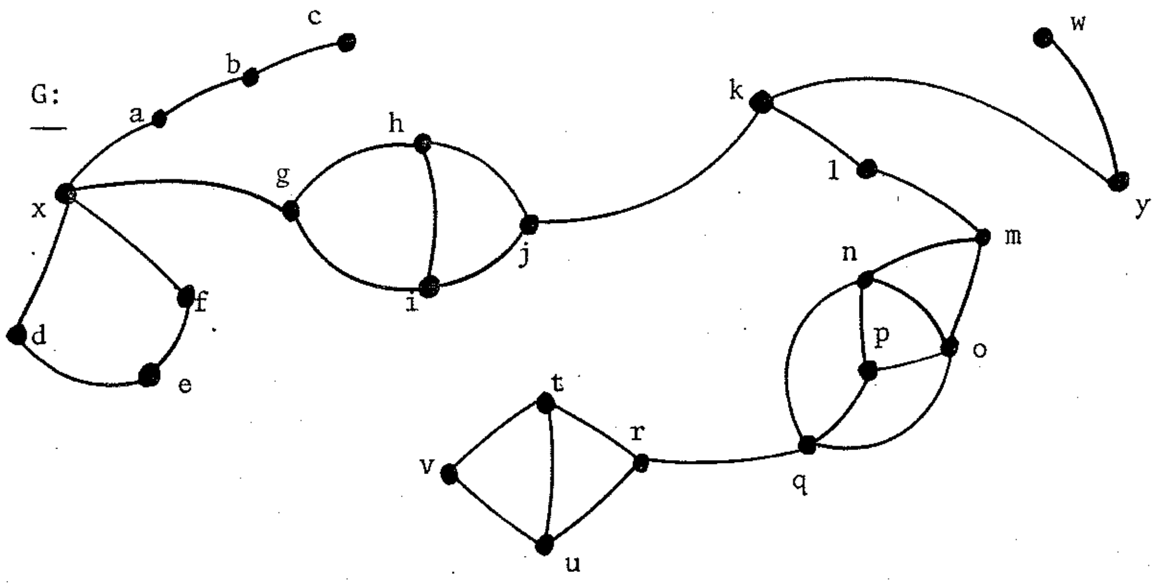
An isolated cycle is a compositional chain one of whose nodes is a cut-point. The chain is directed (arbitrarily) so that it closes a directed cycle. The line from the cycle that leads into the cut-point, say q, is replaced by a line directed to a newly generated node q'. The line (q,q') is added to the graph. Finally, for all p such that $(q,p) \in G''$, the line (q,p) is replaced by the line (q';p).

STEP V)

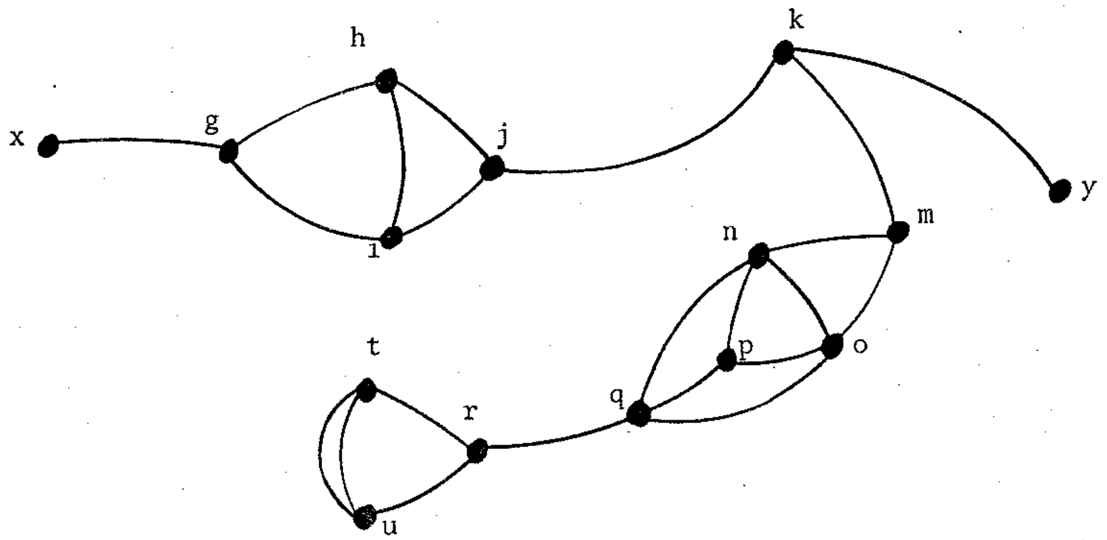
Nodes with indegree 0 (including any node with degree 1), say q, other than the source, result in the directed line U(s,q).

Before examining the properties of this algorithm, we shall give a single comprehensive example of how it operates. We show the graph below in all of its forms: G, G', G'', and the final form G'''.

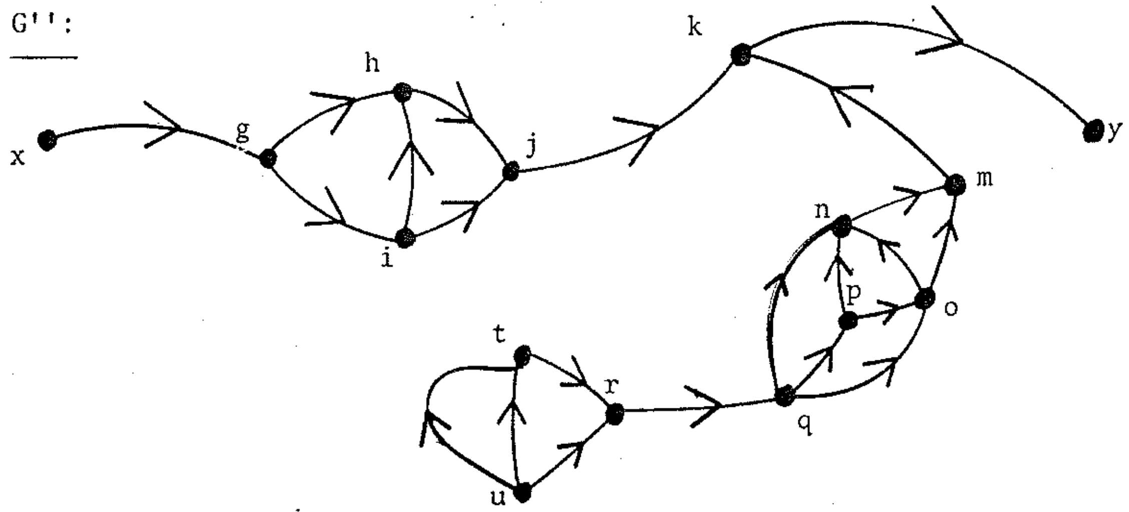
<ins>G:</ins>



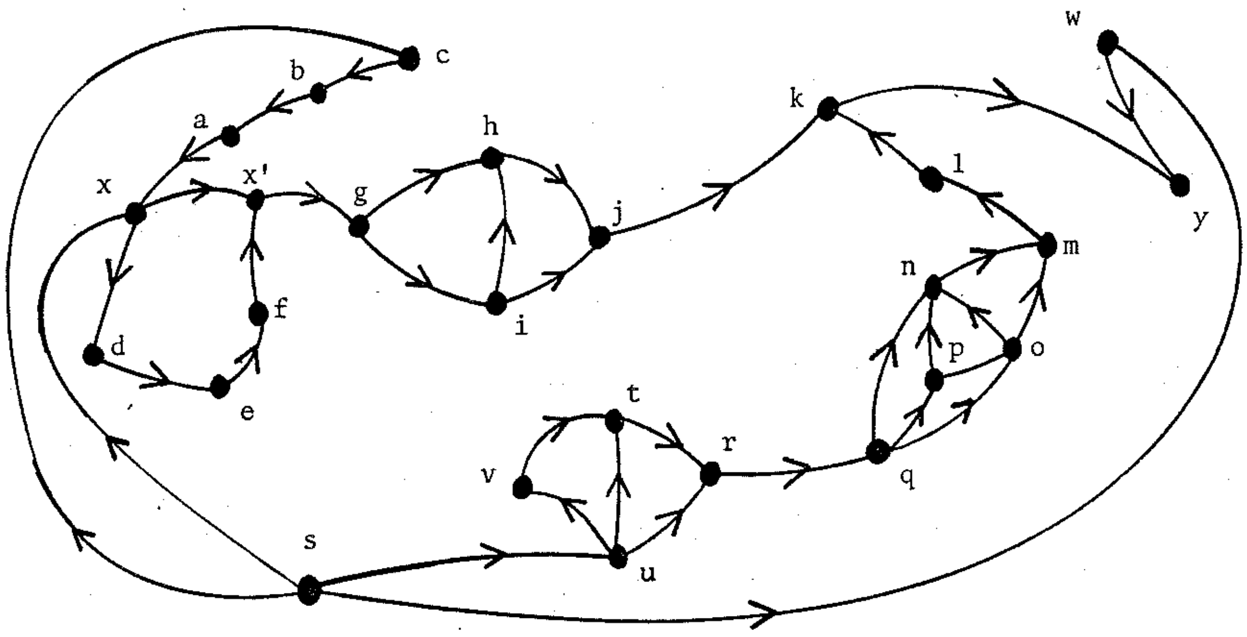
<ins>G':</ins>



G'':



G''':



6.2 ON THE VALIDITY OF THE ALGORITHM

We shall now investigate the properties and utility of the algorithm, and demonstrate that it does in fact parse our language correctly.

LEMMA 6.1: The graph G'' is acyclic.

PROOF: Suppose there is a cycle $C = g_1, g_2, \ldots, g_n, g_1$ in G''. Though it was only made explicit in step (IIa), at all times what the algorithm was actually doing was computing f(g) for each g in G', and then directing the line (p,q) from p to q if and only if $p \succ q$. Clearly the function f is unique in the sense that for $p \neq q$, $f(p) \neq f(q)$. Further the relation $\succ$ is irreflexive, antisymmetric, and transitive. But the presumed cycle C indicates that $g_1 \succ g_n \succ g_1$ and hence by transitivity $g_1 \succ g_1$, contradicting the irreflexiveness of $\succ$.

LEMMA 6.2: The graph G''' is weakly connected.

PROOF: Step (II) checks for the connectedness of G', and none of the transformations is capable of disconnecting it.

LEMMA 6.3: For each node $g \in G''$ other than the source and the sink, $ID(g) > 0$ and $OD(g) > 0$.

PROOF: Step (V) explicitly ensures that no intermediate node has indegree 0. Every intermediate node has positive outdegree for we have N(G') equal to the union of the Y_i and thus for each node g, there is a k such that $g \in Y_k$, which by definition of Y_k implies that the directed line $(g,p) \in G''$ for some $p \in Y_{k-1}$ (and of course for $g \neq y$ we must have $k \geq 1$).

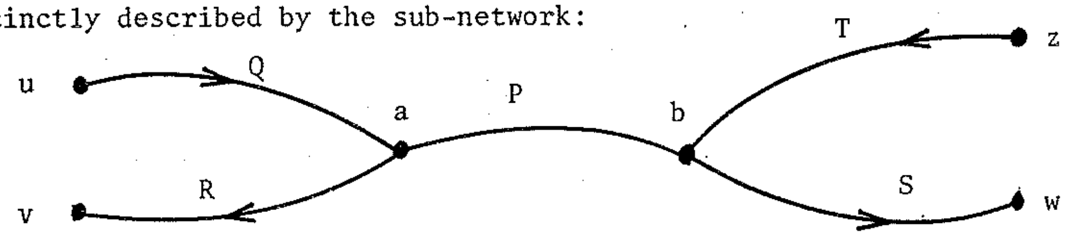
LEMMA 6.4: The network G''' is N-connected. That is every node lies on a directed path from source to sink.

PROOF: Suppose that some point g does not lie on a directed path to y. Then neither does the point p for $(q,p) \in G'''$ (which we are assured exists by lemma 6.3). We repeat this argument for p and so on, indefinitely. But since our graph is finite, this implies that there is a cycle, contradicting lemma 6.1. Likewise, the same argument convinces us that every intermediate node lies on a directed path from the source.

We have given little rationale for our algorithm, particularly rule II$_a$ seems quite arbitrary. We have:

LEMMA 6.5: Given two undirected, isomorphic networks H and K, and directing the lines of H in *any arbitrary way* that results in H being N-connected, and directing the lines of K in *any arbitrary way* subject to the same condition, the two directed networks H' and K' represent, under the relational interpretation of section 6.1.1, identically the same relation.

PROOF: We need only consider networks in which all intermediate nodes have degree ≥ 3. This is because G'' has this property, but more to the point, if a node has degree 2 and the network is N-connected, then altering either one of the lines and not the other would violate the N-connectedness; both lines would have to change together, so we might as well consider them as one line. Further, N-connectedness means that no intermediate node can have degree < 2. That is, for H' and K' to be N-connected, we need only worry about their abstractions as produced by step (I) of the algorithm. Now let H be a directed network that is N-connected and let K be an exact copy of H, save that the line (b,a) has been inverted to be (a,b). In order that K be N-connected we must have in H that $OD(b) \geq 2$ and $ID(a) \geq 2$. The relational interpretation in question would "collapse" multiple arcs coming into or out from a node by treating such multiple arcs as a single complex relation (analogous to a "sub-expression" in an algebraic compiler). Thus the situation is succinctly described by the sub-network:



We know nothing about the nodes u, v, w and z except that u and z cannot be sinks and v and w cannot be the source; we only assume that the labeled lines Q, R, S and T exist (each one may represent a complex sub-relation) and are so directed. In H the relation P is directed b to a; in K, P is directed a to b. We now write out in predicate calculus exactly what the two relational interpretations of these two network portions would be:

H:

$(x)(y)(\exists z)(\exists b)[X(x,z) \wedge T(z,b) \wedge (\exists w)(S(b,w) \wedge (\exists a)(\exists u)[X(x,u) \wedge$
$Q(u,a) \wedge P(a,b) \wedge (\exists v)(R(a,v))])]$

K:

$(x)(y)(\exists u)(\exists a)[X(x,u) \wedge Q(u,a) \wedge (\exists v)(R(a,v) \wedge (\exists b)(\exists z)[X(x,z) \wedge$
$T(z,b) \wedge P(a,b) \wedge (\exists w)(S(b,w))])]$

In the above we have tacitly assumed that all flow is from source to sink so that we may introduce the dummy relation X. We next reduce the two wffs, above, to prenex normal form:

H:

$(x)(y)(\exists z)(\exists b)(\exists w)(\exists a)(\exists u)(\exists v)[X(x,z) \wedge T(z,b) \wedge S(b,w) \wedge X(x,u) \wedge$
$Q(u,a) \wedge P(a,b) \wedge R(a,v)]$

K:

$(x)(y)(\exists u)(\exists a)(\exists v)(\exists b)(\exists z)(\exists w)[X(x,u) \wedge Q(u,a) \wedge R(a,v) \wedge X(x,z) \wedge$
$T(z,b) \wedge P(a,b) \wedge S(b,w)]$

Now the matrices of the two wffs are identical up to order, but order is immaterial for conjunction is commutative. Also in the prefix, the only difference is in the order. Logically this is indeed a difference, but it is not relevant here. In particular, if we next skolemize our formulas, they become identical, save for the arbitrary naming of the skolem functions. The reason why the order of quantification is immaterial is that if we have $(\exists p)(\exists q)$, then, strictly speaking, the q asserted to exist is a function of our *choice* of p asserted to exist. In the present context, however, we do not choose one p that satisfies the wff, rather we are always gathering an answer *set* and we always use all p's that satisfy the wff. This consideration takes away any meaning from the order of quantification.

The result now follows by a simple inductive argument.

Another way of intuitively seeing why this lemma is true is to notice that inverting a line means (assuming N-connectedness) that instead of having two lines going out, we have two coming in, or vice-versa. The thing to recognize is that in either case the interpretation is *intersection*, and thus the direction is immaterial.

We now have:

THEOREM: The algorithm presented in section 6.1.3 for converting the graph G to the network G''' terminates, and when it does the lines have been directed so as to exactly represent the meaning of the sentence under the relational interpretation.

The correctness of the algorithm has just been proven in the preceeding lemmas. Termination is guaranteed by the finiteness of G and the fact that there is no back-up in the sense that no node is ever considered more than once (note that the Y_i computed by Step II are disjoint).

6.3 FURTHER SEMANTICS

Phase II having converted G to G''' then gives way to phase III to convert G''' into a program template. This is extremely straightforward, and involves starting at the sink, y, and forming the indicated compositions and intersections, walking backwards through the network to the source.

After this has been done, phase III checks on the semantics of the definition. First of all, circular definitions are not valid since they would lead to an infinite recursion when the program was executed. This is checked by phase III using the list of relations that the present relation is defined in terms of (see Fig. 6). Each name on that list also has a "circularity" list (empty if the name is not a defined relation) which is appended to the current list. (A recursive definition does not place itself on its own circularity list.) The list is processed by comparing each name on it with the name of the relation being defined: if the same then reject the definition as being circular; if not get the new list to add, if any. This check must terminate by exhausting the list, since none of the relations on the list is circular.

The relation is checked for symmetry and transitivity. Any clause in the definition of R that consists only of R either has the arguments in the same order [R(x,y)], in which case that clause is deleted as carrying no information, or else the arguments are reversed [R(y,x)], indicating symmetry. Symmetry is flagged (section 5.1) and that clause produces no code. Transitivity is recognized when a clause consists of the relation, say R, being composed with itself, and that being the entire clause. Again this is flagged and no code generated. An interesting situation arises when we have:

R(x,y) = R(x,a) .A. R(y,a)

which is the expanded form of

R = R / .CON. R

We have the following relational proposition:

PROPOSITION: If the relation R satisfies $R / R^{-1} \subseteq R$, then

a) if R is reflexive, then R is an equivalence relation.

b) if R is symmetric, then R is transitive.

c) if the domain and range of R coincide, then R is reflexive, and hence an equivalence relation.

An example of such a situation is the definition of SIBLING on page 77. Since it is impossible at compile time to know anything about the domain or range of a relation, (c) cannot be used (although it might be a very useful "guess" that they are in fact identical and R is an equivalence relation). But in the definition of SIBLING on page 77, we do know that it is symmetric allowing (b) to be applied.

The last piece of semantics detected by phase III that is of general interest, is the case where the relation R is defined as being the composition of itself with something else, which we might prototypically designate by:

R = P / R / Q

This definition is recursive and expanding it recursively we have:

$R = P/R/Q = P/(P/R/Q)/Q = P/(P/(P/R/Q)/Q)/Q = \ldots$

indicating that:

$$R = P^c \,/\, R \,/\, Q^c$$

where P^c denotes the *transitive closure* of P. An example of this is illustrated on page 77 (PARENT = ... PARENT/SPOUSE) and the resulting transitive closure of SPOUSE shown being performed by the @*trn* function on page 79.

One final observation is the invalidity of disconnected graphs. One might well want to define e.g.

$$R(x,y) = Q(x,a) \text{ .A. } P(b,y)$$

which specifies that the domain of R is the domain of Q and the range of R is the range of P, and each domain element is related by R to each range element. This is well-defined. However, it is not a very useful relation and the Q and P of the example are actually being used as unary predicates—or *properties*, i.e. x lies in the domain of R and is related to the entire range of R if and only if x has the property of being a domain element of Q. The fact that TRAMP does not allow such definitions is not considered to be a handicap, except perhaps theoretically.

Proof of the relational proposition:

Basic Hypothesis: $R \,/\, R^{-1} \subseteq R$ (6.7)

Starting with (b), R is symmetric means $R = R^{-1}$, and the basic hypothesis (6.7) says:

$$(x,y) \in R \wedge (y,z) \in R^{-1} \implies (x,z) \in R$$

Since $R = R^{-1}$, we can rewrite this as

$$(x,y) \in R \wedge (y,z) \in R \implies (x,z) \in R$$

which says that R is transitive.

Part (a) requires that we show reflexiveness implies symmetry. If R is reflexive, then

$$(x,y) \in R \implies (x,x) \in R \wedge (y,y) \in R$$

thus for (x,y) R, we have

$$(y,y) \in R \wedge (y,x) \in R^{-1}$$

which implies by (6.7) that $(y,x) \in R$, and thus R is symmetric. By part (b), R is also transitive, hence an equivalence relation.

For part (c) we note that $(x,y) \in R \implies (y,x) \in R^{-1}$, and thus by (6.7) we have $(x,y) \in R \implies (x,x) \in R$. That is, the domain of R is a subset of the range of R, and each domain element is in relation R to itself. If the domain is not a proper subset, then the domain and range coincide and we have each range element standing in relation R to itself, and thus R is reflexive. By part (a), R is then an equivalence relation.

Chapter 7: EXAMPLES AND CONCLUSIONS

7.1 QUESTION-ANSWERING

At the outset we placed TRAMP in the framework of "question-answering." The power of the language as pertains to question-answering appears to be twofold: the associative storage structure is decidedly question-oriented and the inference capabilities achieved by facilitating *intensional* definitions of relations. The inference mechanism is by no means as general or formal as a resolution theorem-prover, but in some ways it is more realistic and useful. In particular, one has some facility at least for incorporating semantics into the system (in the sense that the context surrounding a relation may sometimes be used instead of having to axiomatize all predicates which leads to the problems illustrated below)—resolution must operate completely on syntax. Further, since TRAMP resides in a "text-processing" environment provided by TRAC, a pleasant interface with the question-asker is accomplished relatively easily.

Before presenting an example of a TRAMP-coded question answering program, we would like to emphasize that TRAMP is in no way a fully adequate tool for constructing really intelligent question-answerers. The system does present a single method of deduction that unlike resolution is *not* complete, but does some things more efficiently and in a more natural way. In Chapter 2 we tried to point out that the resolution approach to question-answering, by itself, was inadequate. Certainly our approach, in isolation, is inadequate. We think that in the years ahead a corpus of knowledge will develop on various strategies of deduction, none completely utilitarian, all of some value, out of which eventually the type of question-answerer envisioned by Turing [34] might be constructed.

The program listing on the following pages is intended only as an illustration of how one might write a program having a pleasant "front-end" interface, and possessing enough power to perform respectably on Cooper's now standard chemistry example, as well as handle virtually all of the kinship examples presented earlier. The program accepts an "English-like" language L, of which a BNF description is depicted in Figure 7. The program accepts Cooper's original English statements, whereas Green states [2] that two hours were required to perform the translations from English to predicate calculus by hand. Indeed, it took only slightly longer than two hours to write the program shown below.

FIGURE 7: B N F Grammar for the Language L

```
<statement>   :== <fact>. | <rule>. | <question>?
<fact>        :== SYNONYM: <id> <noise> <id> | <idl> <is> <q> <art> <idl>
                  | <id> <rel> | <idl> <rel> <idl> | <id>'S <rel> IS <id>
                  | <id> IS <art> <rel> OF <id> |  THE <rel> OF <id> IS <id>
<article>     :==  A | AN | ANY | EITHER | BOTH
<art>         :==  <article> | <art> <art> | THE | λ
<rel>         :==  <id>
<q>           :==   NOT | λ
<is>          :==   IS | ARE
<idl>         :==  <id> | <id> , <idl> | <idl> AND <id>
<id>          :==  <alphanumeric string>
<noise>       :==  <id> | <art> | <is> | <noise> <noise>
<rule>        :==   IF <exp> THEN <rel> | NO <rel><is> <art> <rel>.
                  |  <article> <rel> IS <art> <exp>
<exp>         :==  <rel> | <exp> <op> <art> <exp> | TRANSITIVE
                  |  <rel>'S  CONVERSE | THE CONVERSE OF <rel>
                  |  <exp> OF <art> <rel> | <exp>'S <rel>
<op>          :==   AND | OR
<question>    :==  DID <id> <rel> <id> | WHO <is> <id>'S <rel>
                   |  WHAT <noiz> <art> <id> <noise> <rel>
                   |  HOW MANY <id>S <noiz> <id> <rel>
                   |  <qual> <id> <art> <id > |<is> <id> NOT <art> <id>
                   |  <is> <id> <art> <idx>
<noiz>        :==  <is> | <id>
<qual>        :==  IS EVERY | ARE ALL | <is> NO |<is> SOME
<idx>         :==  <id> | <id> , <idl> | <idl> <that>
<that>        :==  THAT <is> <idl> | THAT <idl>
```

(λ represents the null string.)

```
(1)    #(DS,START,(#(DS,READ,##(RS))#(SS,READ,(, ),:,.)#(DS,READ,##(READ,;))
               #(#(LS))#(START)))'

(2)    #(DR,LEXTYPE,IS;ARE,VERB)#(DR,LEXTYPE,AND;OR,OP)
       #(DR,LEXTYPE,THE;EITHER;BOTH;A;AN;ANY,ART)
       #(DR,LEXTYPE,CONVERSE;TRANSITIVE;SYNONYM;IF;THEN;OF;WHAT;WHO;DID;HOW;
               NO;NOT;SOME;EVERY;ALL;THAT,KEYWORD)'

       #(SET,GET,RL)#(DS,RL,(#(GET,#(SYN,A),#(SYN,O),#(SYN,V))))#(SS,RL,A,O,V)
       #(DS,SYN,(##(GET,SYN,X,**);X))#(SS,SYN,X)
       #(DDR,IS = SYN .V. .CON. IS .V. @IS)'

(3)    #(DS,LS,(#(DS,@A,##(READ))#(DS,@@,##(READ))#(SS,@A, ,?)#(SS,@@, ,?)
               #(LS1)))
       #(DS,LS1,(#(DS,@B,##(CS,@A))#(EQ,##(@B),,(##(SRCH,##(READ),?,QUERY,
               INPUT)),(#(EQ,##(GET,LEXTYPE,##(@B),**),,(#(DR,LEXTYPE,##(@B),
               ID)))#(LS1)))))'

(4)    #(DS,INPUT,(#(DS,@0,##(CS,@@))#(EQ,##(@0),THE,(#(THE)),(#(GET,LEXTYPE,
               ##(@0),*@A*)#(EQ,##(@A),KEYWORD,(#(KEY)),(#(EQ,##(@A),ID,
               (#(FACT)),(#(RULE)...)'
       #(DS,QUERY,(#(SS,READ, AND )#(DS,@@,##(READ,;))#(SS,@@, ,?)#(COUNT)
               #(##(@1))))'
       #(DS,COUNT,(#(DS,CX,0)#(CR,@@)#(CONT)))
       #(DS,CONT,(#(DS,@A,##(CS,@@))#(EQ,##(@A),,,(#(EQ,##(GET,LEXTYPE,##(@A),
               **),ART,,(#(DS,CX,#(AD,##(CX),1))#(DS,@##(CX),##(@A))))#(CONT)))))'

(5)    #(DS,THE,(#(COUNT)#(DR,##(@1),##(@3),##(@5))))
       #(DS,KEY,(#(COUNT)#(##(@1))))
       #(DS,RULE,(#(COUNT)#(DDR,##(@1)=#(EXP,3))))
       #(DS,SYNONYM,(#(DR,SYN,##(@2),##(@##(CX)))))
       #(DS,IF,(#(DDR,##(@##(CX))=#(EXP,2))))
       #(DS,NO,(#(DDR,##(@2)=.N.##(@##(CX)))))'
```

```
(6)   #(DS,FACT,(#(SS,READ, AND )#(DS,@@,##(READ,;))#(SS,@@, ,.)#(COUNT)
              #(EQ,##(GET,LEXTYPE,##(@2),**),VERB,(#(VERB)),(#(SRCH,##(@1),
              ('S),(#(SS,@1,('S))#(DR,##(@2),##(@1),##(@4))),(#(EQ,##(CX),
              2,(#(DR,PROP,##(@1),##(@2))),(#(DR,##(@2),##(@1),##(@3)...)'
      #(DS,VERB,(#(SRCH,##(@@, ), OF ,(#(DR,#(NEXT),##(@##(CX)),##(@1))),
              (#(SRCH,##(@@, ), IS A ,(#(VERB1)),(#(SRCH,##(@@, ), IS AN ,
              (#(VERB1)),(#(EQ,##(@2),ARE,(#(VERB2)),(#(DR,@IS,##(@1),
              ##(@3)...)'
      #(DS,VERB1,(#(RELATE,##(@3),##(@1))#(DR,SUB,##(@3),##(@1)...)
      #(DS,RELATE,(#(DS,@A,X)#(DS,@B,Y)#(SS,@A,;)#(SS,@B,;)#(RELAT)))
      #(DS,RELAT,(#(DS,@C,##(CS,@A))#(EQ,##(@C),,,(#(DDR,##(@C)=
              ##(@B,.V.))#(RELAT)...)#(SS,RELATE,X,Y)'

      #(DS,VERB2,(#(NL,#(LAST,@1))#(EQ,#(LAST,@3,S,(#(VERB1)),(#(DR,@IS,
              ##(@1),##(@3)...)'
      #(DS,NEXT,(#(DS,@E,##(CS,#(CR,@@)@@))#(NEX)))
      #(DS,NEX,(#(DS,@E,##(CS,@@))#(EQ,##(GET,LEXTYPE,##(@E),**),ID,(##(@E)
              ),(#(NEX)...)'
      #(DS,LAST,(#(DS,X,#(SS,X,S;)##(X,;))#(NL,#(CS,X))#(DS,@E,##(CN,X,-1))
              #(EQ,##(@E),S,(#(DS,X,#(CN,X,#(SU,#(LEN,##(X)),1)))))##(@E)))
              #(SS,LAST,X)'

(7)   #(DS,EXP,(#(EQ,##(@Y),THEN,,(#(EQ,Y,##(CX),(##(@Y)),(#(EQ,##(@#(AD,
              Y,1)),OF,(#(COMP,#(AD,Y,1))/##(@Y)#(EXP,##(CXX))),(#(SRCH,##(
              @Y),('S),(#(SS,@Y,('S))##(@Y)/#(EXP,#(AD,Y,1))),(#EQ,##(GET,
              LEXTYPE,##(@Y),**),OP,(.#(EQ,##(@Y),OR,V,A).#(EXP,#(AD,Y,1))),
              (#(PS, *** DO NOT UNDERSTAND STATEMENT.)...)#(SS,EXP,Y)
      #(DS,COMP,(#(EQ,##(GET,LEXTYPE,##(@Y),**),OP,(#(DS,CXX,Y)),(#(EQ,##(
              @Y),THEN,(#(DS,CXX,Y)),(#(SRCH,##(@Y),('S),(#(SS,@Y,('S))##(@Y)
              /#(COMP,#(AD,Y,1))),(#(EQ,##(@#(AD,Y,1)),OF,(#(COMP,#(AD,Y,2))
              /##(@Y)),(##(@Y)#(DS,CXX,#(AD,Y,1))#(EQ,Y,##(CX),(#(DS,@##(CXX)
              ,THEN)...)#(SS,COMP,Y)'
```

```
(8)   #(DS,HOW,(#(CT,#(RL,##(@##(CX)),##(@#(SU,##(CX,1)),**))))
      #(DS,WHAT,(#(RL,##(@##(CX)),#(NL,#(CS,@@)#(CS,@@))#(NEX),**)))
      #(DS,DID,(#(EQ,#(RL,##(@3),##(@2),##(@4)),1,##(YES))))
      #(DS,WHO,(#(RL,##(@4),#(SS,@3,('S))##(@3),**)))
      #(DS,IS,(#(EQ,##(GET,LEXTYPE,##(@2),**),KEYWORD,(#(QUAL)),(#(EQ,##(@3),
             NOT,(#(NOT)),(#(SRCH,##(@@, ), THAT IS ,(#(THATIS)),(#(SRCH,
             ##(@@, ), THAT ,(#(THAT)),(#(EQ,#(INT,##(@2),#(RL,IS,##(@3),**)
             ),,#(NAY)...)'
      #(DS,NOT,(#(EQ,#(INT,##(@2),#(RL,IS,##(@4),**)),,#(YES)...)
      #(DS,THATIS,(#(EQ,#(INT,##(@2),#(INT,IS,#(NL,#(CS,@@),#(CS,@@),#(NEX);
             #(NEX),**)),,##(NAY)...)
      #(DS,THAT,(#(EQ,#(INT,##(@2),#(INT,#(RL,IS,##(@3),**),#(RL,PROP,**,
             ##(@##(CX)));#(EQ,##(@#(SU,##(CX),1)),THAT,,(#(RL,##(@#(SU,##(CX)
             ,1)),##(@##(CX)),**))))),,##(NAY)))
      #(DS,QUAL,(#(EQ,##(@2),SOME,(#(SOME)),(#(EQ,##(@2),NO,(#(SOME)),(#(EQ,
             #(SYMD,#(RL,SUB,##(@3),**),#(RL,SUB,##(@4),**)),,##(YES)...)
      #(DS,SOME,(#(EQ,#(INT,#(RL,SUB,##(@3),**),#(RL,SUB,##(@4),**)),,##(NAY)...)
      #(DS,YES,(YES,NO(,) NOT AS FAR AS CAN BE DETERMINED.))
      #(DS,NAY,(NO(,) NOT AS FAR AS  CAN BE DETERMINED,YES))'
```

The program listing on the preceeding pages represents a fairly sophisticated use of UMIST and is almost unreadable even to those familiar with the language (it *writes* quite fluently though). The program is coded in UMIST and not all of the functions used are available in TRAC. We will now briefly describe the program structure and strategy. Note that the program is oversimplified, for example it has only a lexicon for keywords and would not properly parse Cooper's fact: COMBUSTIBLE THINGS BURN, since it doesn't know the meaning of "THINGS." The parenthesized numbers in the margin of the program listing identify program "blocks" which will be referred to in the discussion below.

Block (1) is what amounts to the "main program." It is the procedure which replaces the UMIST "idler" and reads in each statement, standardizes punctuation, calls the lexical scanner (LS), then calls on the appropriate routine (INPUT or QUERY), after which it regains control recursively to read the next input.

Block (2) is the definition of the lexical types of all of the reserved words of the language. Any other word is of type ID. This block also provides the system thesaurus by redefining RL. All calls to the redefined RL will first make any substitutions found in the thesaurus. [Note that this provides a fairly natural way of answering the "nonsense" question of Cooper: IS MAGNESIUM MAGNESIUM, since the question type "IS A B" will see if A is a synonym for B, and by the definition of SYN in block (2), everything is a synonym for itself. This is opposed to the extra equality axioms needed by Green to answer the same question.]

Block (3) is the lexical scanner. It determines the type of statement by whether or not it ends with a question mark, and also ensures that each word in the statement (words are delimited by blanks) is assigned a lexical type. Throughout, the symbol "@" is used as a reserved symbol, and the strings @A, @B, etc. are internal temporary forms used by the program.

Block (4) contains the two procedures that can be called: INPUT and QUERY. Each of these functions performs initializations and assigns each form @1, @2, ... , @n the sequential content word of the statement. Then, on the basis of the first word of the statement, it decides which procedure to call to process that particular statement.

Block (5) contains most of the procedures that are called as a result of the first word being a keyword. The procedure RULE processes rules that do not begin with a keyword but are identified by context. Block (6) shows the various functions necessary to process facts. Although this program does distinguish between rules and facts, it is a different distinction than TRAMP makes. In particular, the routine FACT will generate calls to both *dr* and *ddr*, thus making some rules retrievable.

The heart of all of the various procedures used to parse "rules" is the "expression scanner" (EXP) shown in Block (7). The expression scanner operates recursively to generate the correct arguments to *ddr*. EXP calls on a subroutine COMP to form "reverse" compositions necessitated by constructions of the form "R1 *OF* R2" (the interpretation of such constructions is discussed on page 70).

Finally, block (8) translates questions into the proper retrieval requests.

As an example of how the program operates, suppose that it is input:

```
OXYGEN, SULFUR, COPPER AND IRON ARE ELEMENTS.
```

The main program, START, will remove the commas yielding:

```
OXYGEN;SULFUR;COPPER AND IRON ARE ELEMENTS.
```

Next the lexical scanner assigns:

```
OXYGEN, SULFUR, COPPER, IRON, ELEMENT  type ID
AND type OP
ARE type VERB
```

and statement type FACT.

Since the statement type is FACT, the routine INPUT is called which in this case will just pass the call on to the routine FACT. FACT will change the "AND" to a semi-colon and call COUNT which stores:

```
@1 = OXYGEN;SULFUR;COPPER;IRON
@2 = ARE
@3 = ELEMENT
```

On the basis of the keyword ARE, the routine VERB2 is finally called, which ultimately generates the TRAMP function calls:

```
#(DDR,ELEMENT=OXYGEN.V.SULFUR.V.COPPER.V.IRON)#(DR,SUB,ELEMENT,OXYGEN;
      SULFUR;COPPER;IRON).
```

In lieu of exhibiting output of the program we shall discuss of what significance such output would be. Suffice it to say that the program would handle all of the kinship examples presented earlier, a class of questions that might be asked of an automated library retrieval system, and would handle almost all of Cooper's example (page 127) correctly. It would not understand questions 16 or 17 of Cooper's chemistry example; it would miss question 23 because it would not understand fact #10; and it would expect all twenty-three questions to be worded as questions, not facts. All of the facts (except the last three used by Green) would be accepted exactly as they are.

This "question-answering" program is shown *only* as a demonstration of how TRAMP can be used. Actually, to call that program a question-answerer is somewhat presumptuous. Let us now take a closer look at the area termed "question-answering" and examine the validity of the results being obtained therein.

All too often in artificial intelligence one is presented with a "machine" that purportedly performs some task, e.g. answering questions. Too frequently the significance of such a machine is appraised before its insides are examined: "Here is a black box; this is its input and that is its output." If we do not closely inspect this machine, but simply accept it as a "black box," we may well be the victims of a hoax. If the machine has all of its responses built-in, then we should not want to call it heuristic or say that it is a very interesting machine simply on the basis of its responses.

Inasmuch as Cooper [3] essentially made the first organized attempt at the general problem of question-answering, his chemistry example has become a standard of comparison and point of reference. For example Cooper, Slagle [35], Green, among others, run their

SIMPLE CHEMISTRY EXAMPLE

BASIC FACTS

1. magnesium is a metal
2. magnesium burns rapidly
3. magnesium oxide is a white metallic oxide
4. oxygen is a nonmetal
5. ferrous sulfide is a dark-gray compound that is brittle
6. iron is a metal
7. sulfur is a nonmetal
8. gasoline is a fuel
9. gasoline is combustible
10. combustible things burn
11. fuels are combustible
12. ice is a solid
13. steam is a gas
14. magnesium is an element
15. iron is an element
16. sulfur is an element
17. oxygen is an element
18. nitrogen is an element
19. hydrogen is an element
20. carbon is an element
21. copper is an element
22. salt is a compound
23. sugar is a compound
24. water is a compound
25. sulfuric acid is a compound
26. elements are not compounds
27. salt is sodium chloride
28. sodium chloride is salt
29. oxides are compounds

EXTRA FACTS USED BY COOPER (AND GREEN)

30. metals are metallic
31. no metal is a nonmetal
32. dark-gray things are not white
33. a solid is not a gas
34. any thing that burns rapidly burns

MORE EXTRA FACTS USED BY GREEN

35. ferrous sulfide is a sulfide
36. equality is reflexive
37. equality is symmetric
38. equals can be substituted for equals

QUESTIONS

1. magnesium is a metal
2. magnesium is not a metal
3. magnesium is a nonmetal
4. magnesium is not a nonmetal
5. magnesium is a metal that burns rapidly
6. magnesium is magnesium
7. some oxides are white
8. no oxide is white
9. oxides are not white
10. magnesium oxide is an oxide
11. every oxide is an oxide
12. ferrous sulfide is dark gray
13. ferrous sulfide is a brittle compound
14. ferrous sulfide is not brittle
15. some sulfides are brittle
16. ferrous sulfide is not a compound that is not dark gray
17. anything that is not a compound is not ferrous sulfide
18. no dark gray thing is a sulfide
19. ferrous sulfide is white
20. sodium chloride is a compound
21. salt is an element
22. sodium chloride is an element
23. gasoline is a fuel that burns

question-answering programs on this chemistry example as some sort of demonstration that it works. But this chemistry example involves almost *no* deductive logic; rather it is nearly entirely a semantic game. Typical "questions" are:

MAGNESIUM IS MAGNESIUM, and EVERY OXIDE IS AN OXIDE.

The full example is shown on page 127. Of the twenty-three questions, about the deepest deduction required is to go from:
GASOLINE IS A FUEL, GASOLINE IS COMBUSTIBLE, COMBUSTIBLE THINGS BURN.
to: GASOLINE IS A FUEL THAT BURNS.

The questions in the chemistry example involve being able to translate from English and comprehend the statement mechanically, not performing logical deduction. In this light, one wonders why Green should have performed *all* of the translation by hand (requiring two hours). Cooper was able to correctly answer all questions except #19, 20, 22 and 23. One also wonders how he was able to answer question 15 without fact 35. Another discrepancy lies with fact #38 used by Green. How could that sentence possibly be translated into first-order logic? Of course it cannot, and so the single instance of that schema that will be required is translated instead, viz.

(FA(X Y)(IMP(AND(IS X Y)(COMPOUND X))(COMPOUND Y))).

How fortunate that the translator could foresee how this fact would be needed. A final observation from a cursory scan of page 127, is that facts 36 and 37 express only the properties of equality required for this example—no need to say that equality is an equivalence relation since this example does not use the transitivity of equality.

It would have been considerably easier to write a TRAMP program specifically for this example that would have answered all the questions correctly, than the program for the language L. It would have been easier still to have just translated all of the facts and questions into TRAMP code directly. However, the problem involved is really one in mechanical translation, which we said at the outset was not the topic of this paper—we are interested in methods of deduction.

The point about the chemistry example is that it does not demonstrate deductive capabilities, yet that is what people use the example to illustrate. Our model of question-answering, explicitly employed by Green, assumes that the information is in a canonical form. But if these questions are phrased canonically they become quite trivial, even absurd to the point of nonsense, or else unanswerable! For example question 15 is unanswerable from the original 29 (or 34) facts, since we do not know what a sulfide is.

An example of how a program can be adapted to produce desired responses is provided by question 17. One first becomes suspicious of this question by looking at the dialogue that resolution produced:

Q ANYTHING THAT IS NOT A COMPOUND IS NOT FERROUS SULFIDE

A YES, X = MA

We have a strangely worded "question" which was worded in that way as a challenge to a mechanical parser, and we get back the answer "YES," the validity of which is not immediately clear. The added information that "X = MA" only helps to confuse the issue. This question seems extremely simple requiring the deduction that

ANYTHING THAT IS NOT A COMPOUND IS NOT FERROUS SULFIDE

from the fact that

FERROUS SULFIDE IS A ... COMPOUND.

Simple as it is, the deduction does require proper representation of the verb "is" by the predicate "IS," which is beyond a straightforward translation to predicate calculus as needed for resolution. The antecedent is translated: (COMPOUND FES), and the question is translated: (FA (Y)(IMP (NOT (COMPOUND Y))(NOT (IS Y FES)))). When the question is negated and placed in skolemized conjunctive normal form, no contradiction can be inferred because the semantics of "is" has not been captured by this translation! This is gotten around by changing the question slightly to require that "something" is not a compound: (AND(EX(X)(NOT(COMPOUND X)))(FA(Y)(IMP(NOT(COMPOUND Y))(NOT(IS Y FES))))). [the abbreviation "FES" also gets us around the nasty problem of knowing that "ferrous" is a kind of sulfide and that "ferrous sulfide" is one entity]. This seems quite legitimate since if the added condition were false, the implication would be vacuously true. We see that it is only a ploy because it is only used when necessary. Implications which can be proven or disproven without requiring that the domain be non-empty do not require that it be non-empty!

The difficulty with representation of English statements in the predicate calculus is not in the direct translation (e.g. as performed mechanically with moderate success in [40]), but in supplying the proper axioms to adequately reflect the meaning of a predicate and its interaction with other predicates. Further ramifications and examples of this problem are discussed at the end of section 7.2.

To further illustrate the semantic problems underlying the chemistry example, from facts # 3 and 5 one should be able to deduce something about the *colors* of magnesium oxide and ferrous sulfide. Yet no system without a good bit of built-in information could answer the question:

WHAT COLOR IS FERROUS SULFIDE?

or phrased as a true/false question:

FERROUS SULFIDE IS COLORED CANARY YELLOW.

Yet, just this kind of information was needed, and perhaps built-in subconsciously by Cooper, to answer question 15 without explicitly knowing that FERROUS SULFIDE IS A SULFIDE.

7.2 EVALUATION AND CONCLUSIONS

We have presented a relational language and implied that while falling short of the expressive power of the predicate calculus, it is a relatively rich language. Exactly what can and what cannot be phrased in this language? As demonstrated in Chapter 6, one can express any relation using the rules defined in Chapter 4: i.e. on the next iteration we can say that any relation that can be represented as the union of *relational networks* of other relations. Still, this is not too helpful.

The language clearly does not have the power of first-order logic, since it does not allow explicit quantification. Also it presently allows only binary and unary relations (though this, theoretically, is not a real problem, and, practically, the restriction could quite easily be lifted). The basic framework underlying the relations is the concept of the set of ordered pairs. Within this framework we can then perform various set operations, constituting a "complete" set of operations, viz. negation, union, intersection, subtraction. Theoretically, negation and union alone suffice, except that for our context negation is ill-defined. The complement of a set is taken with respect to something, and that something is ill-defined in TRAMP. Indeed, this is the most serious, if not the only major restriction in our relational language.

Let us examine the cause and implications of this restriction. In one version of TRAMP "global" complements are not allowed such as specified by the relational sentence:

$$R = .N.\ Q$$

The rationale for that was the relatively inefficient processing that would be required and the uselessness of such constructions. For example, if one defines

$$MALE(x) = .N.\ FEMALE(x)$$

one would have *everything* that is not a female being a male, including such things as cars, stone, rocks, computers, etc. Every single time that any piece of information was added or deleted from the data base, either the relation MALE or the relation FEMALE would change! What one really means is:

MALE(x) = .N. FEMALE(x) .A. (PERSON(x) .V. SEX(x,something))

i.e. one really means that the negation is to be relative to some universe of discourse. TRAMP generally requires that such a universe be specified (as does Levien [33] who only allows the operator AND NOT, the operator NOT by itself being undefined). A later version of TRAMP allows global complements, but does not take action on them until a universe is otherwise defined. Thus R = .N. Q by itself is not used as information about the relation R. But if also we have some definition R = P, then TRAMP will form the relative complement of Q and P, i.e. it will compute the relation P as specified and then subtract out the relation Q.

This problem with complements points out one of the major restrictions of the relational language , viz. it is quite difficult to translate some simple *implications*. One can do this in the framework of set theory, much as Cooper did using Aristotelian logic. To see the difficulty, if we have R = Q, then the meaning of this in TRAMP is "Q IS A SUBSET OF R." Thus we can translate *positive* implications:

$$(x)[Q(x) \supset R(x)]$$

in a straightforward way by translating the sentence into set theory, and saying that Q is a subset of R. But suppose we have the equally simple implication:

$$(x)[Q(x) \supset \sim R(x)]$$

Now, having no convenient way to deal with global complements, this cannot be straightforwardly translated!

The implication $Q \supset \sim R$ is a *negative* statement! In TRAMP we are always looking for "inference rules" that tell us how to derive one relation from others; negative rules do not tell us how to do that, except in the sense of forming a relative complement, since the above negative implication does tell us that Q and R are disjoint so that we may use all of our information to compute a set of Q's and then subtract out any R's that were also computed in the process. But still, even though TRAMP can in fact use the information, it does not directly translate into set theory that we can use, it only tells us that two sets are disjoint, which is a negative statement. It certainly cannot be handled in TRAMP in the same simple way that positive implications are handled.

The relative strengths and weaknesses of the TRAMP relational language are perhaps best brought out by comparison with other languages and methods of deduction. We will start with Elliott [8]. Firstly, his language, GRAIS, did not allow recursive definitions. He sharply distinguished between "primitive" and "composite" relations. A composite relation (one that is defined in terms of primitives) may be defined *only* in terms of primitive relations. The properties, such as symmetric, transitive, etc. may only be applied to primitive relations. Another major difference between GRAIS and TRAMP is the context in which his language is used. Although GRAIS does allow the retrieval of answers, if the relation in question is composite, then one can only receive a YES or a NO. In this context he does not compile the relational sentence, as TRAMP does, but simply interprets it at execution time, by going through and assigning truth values to each relation, thereby reducing the sentence to a proposition with a truth value, and that truth value is the answer to the question. This context simplifies GRAIS processing significantly, and yet that truth value evaluation mechanism is the most complex and involved part of the GRAIS program, indicating that our results in Chapter 6 may be of value.

Levien's work [33] is more recent, and more difficult to evaluate. His language, like Elliott's, does not allow recursive definitions. Also, he too has his relational language operating in a different context than TRAMP. Namely, the relational sentences of his system are part of a program that must be written each time that it is executed; it is not saved by the system as a "rule of inference" as is done in TRAMP. A major difference is that his relations are not very general or abstract, and in fact seem to be built-in to the language by the system for the user. Levien's language is difficult to evaluate since he never precisely defines it, but rather seems content to show that the very simple sentences that he expects can be processed without difficulty. TRAMP has taken the position that anything that is syntactically valid can (and will) arise, and has seen to it that any such sentence can be correctly parsed. Levien does not say how he does process these simple sentences, only that not much attention has been paid to it and the method is ad hoc. It is interesting to note that Levien does not allow global complements either, but requires (syntactically) that a universe be specified. In summary, his language is imprecise and he in no way indicates that pathological sentences may occur, and what action he will take if they do.

It is difficult to compare TRAMP to Green's QA3. First-order logic is certainly richer. He has no parsing problem since he will deal with sentences exactly as they come in (with well-known algorithms to convert to prenex conjunctive normal form). One advantage of TRAMP is the ease with which the relational sentences can be written in English-like sentences to be parsed by a TRAMP procedure, not some hypothetical front-end. Also, though TRAMP has quite the syntactic rigidity of QA3, there is the possibility of incorporating semantics into the triples, whereas this is all but impossible in resolution.

Our approach to fact retrieval is diametrically opposed to that of Green. We cannot claim the theoretical completeness of our system, but we can claim that in practice some things can be done sufficiently easier that it is, overall, more useful for certain applications. The underlying storage structure of TRAMP is perhaps sufficiently better than QA3's storage management that more answers lie within its physical limits, if not its theoretical limits. Certainly we do not discount resolution, and the theorem-proving approach to question-answering, we simply do not think that by itself it is adequate. Neither do we think that the data structure approach by itself is adquate. It seems obvious that later day sophisticated question-answerers will have to draw from various techniques including clever storage management, efficient theorem-proving techniques, and the proper data structure.

An example illustrating the relative strengths of TRAMP and QA3 is Green's dialogue on page 61 of [2] (where he is using a dialogue taken from SIR [36]). On the basis of the two facts:

```
(FA (X)(IF (IN X KEYPUNCH-OPERATOR)(IN X GIRL)))
(FA (Y)(IF (IN Y GIRL)(IN Y PERSON)))
```

QA3 cannot answer the question:

```
(IN KEYPUNCH-OPERATOR PERSON)
```

First of all, one could not directly ask such a question in TRAMP, since TRAMP strictly distinguishes between facts and rules; Green treats them exactly the same—as wffs. However, this question could be asked indirectly in TRAMP and would be answered correctly if phrased correctly. QA3 cannot answer this because a keypunch-operator is not an *element* of the set of persons, it is a *subset*. In TRAMP, almost any representation that allowed that question would answer it since TRAMP deals with sets.

Another example is found on page 62 of [2] consisting of the following dialogue:

```
S   (FA (X Y)(EQV (IS X Y)(IS Y X)))
S   (FA (Y Z W)(IF (AND (IS Y Z)(IS Z W))(IS Y W)))
S   (IN JOHN TEACHER)
S   (IS JOHN JACK)
Q   (IN JACK TEACHER)
  NO PROOF FOUND
CONTINUE
  NO PROOF FOUND
```

QA3 cannot deal with the semantics of the predicate IS in a natural way at all (as earlier illustrated in the chemistry example). The first two statements of the above dialogue attempt to axiomatize the meaning of the predicate IS, but they fail. What is needed in the above example is the statement that EQUALS CAN BE SUBSTITUTED FOR EQUALS, but this transcends first-order logic. This example could be easily done in TRAMP (in fact, the program shown for processing the language L provides two ways of handling this particular situation: firstly by using *ddr* to define JOHN to be equal to JACK, and secondly by using *dr* to make a thesaurus entry), since the formal axiomatization is not required. In summary, QA3 is richer, more formal, and therefore much more rigid. The completeness is purchased at the expense of informality (usefulness) and is of dubious value since it cannot be realized on present day equipment for any useful class of wffs.

Chapter 8: SUMMARY

Let us now review what we have done and what has been learned from it. TRAMP is a language. The TRAMP language consists of two separate sub-languages: the relational language and the associative language. The project initially was undertaken as an experiment in associative memories*. The associative experiment was a failure in the sense that it failed to accomplish the vague notions that had originally motivated it. This motivation was the desire for a "dynamic content-addressable" memory structure. It is difficult to articulate what it was that was wanted. Whatever that was, associative memories seemed like something to try, but they were not the answer.

Since the project was experimental, the implementation was never very polished and was never released to the general public. Nevertheless there were several key users of the implementation whose applications covered a surprisingly wide range. The language was employed for such diverse tasks as: a large scale interactive graphics system [31]; a smaller interactive graphics system using machine-machine communication [32]; artificial intelligence [9]; a project accounting system doing mostly *numerical* calculations, among others.

*At that time the only other experiment was Feldman's [18], which we were not acquainted with until the project was about to begin.

Because of the diversity of the applications to which the TRAMP language has been put, and its successes in such applications, the associative experiment was hardly a total failure. The associative language was designed and implemented rather hastily and it was never returned to, as the relational language was of primary interest—and, of course, the basic problems of deduction that the relational language was designed to alleviate. As such, the only serious deficiency in the associative language is the inability to refer to an associative triple within an associative triple. For example, the sentence:

JOHN GAVE $100 TO MARY

should be expressed in triples as:

S1: GAVE, JOHN, S2

S2: TO, MARY, $100

This would be accomplished by allowing names for associative sentences, or otherwise allowing an entire sentence to be refered to within an association.

In addition to the associative language, a relational data structure and language was provided, requiring a compiler, macro-expander, and several internal utility functions. This relational mechanism and language were of paramount interest. Today there seem to be a great many languages being proposed along the lines of Elliott [8] and Levien [33]. To our knowledge, all of these languages are either less rich than the TRAMP relational language in expressive power, or else they have not been implemented. Thus, the very simple method of dealing with a relatively rich language, presented in Chapter 6 is perhaps of some practical utility.

Though the TRAMP relational language, as implemented, relies heavily on the associative language as well as on the UMIST host language, the ideas embodied in Chapter 6 are applicable to a very wide class of relational languages. Note that the algorithm of section 6.1 is quite independent of the rest of the system, and only assumes that the basic building blocks of the data structure are binary relations (i.e. triples), regardless of how they are actually stored—associatively or otherwise. This is a standard assumption the merits of which we shall not argue here. To repeat, we feel that the algorithm of section 6.1 may be significant if only because of the recent upsurge in interest in relational languages, particularly in the field of data base management. It seems that many languages are being proposed, and of the ones that this author is aware of, the languages are either imprecise, needlessly restricted, or unimplemented.

The individual constituents of the system were each motivated and explained in some detail. Chapter 5 explained how these various pieces fit together to form the TRAMP language. Chapter 7 analyzed and evaluated the TRAMP language and compared it to other notable related work in the field.

APPENDIX A

A BRIEF SUMMARY OF THE UMIST LANGUAGE

APPENDIX A: A BRIEF SUMMARY OF THE UMIST LANGUAGE

The following excerpts from the Umist manual are reproduced with the kind permission of Mr. Tad Pinkerton. What follows is partial and incomplete and is intended only to familiarize the reader with the structure of the language and enable him to follow the Tramp definitions and examples. A complete description of the Umist language may be found in Vol. II of reference 23.

A level of the TRAC language called "TRAC 64" is described in [15]. It is the basic standard and point of reference for Umist. A good discussion of TRAC 64's design goals and principles is given in reference [37]. Much of the motivation for the development of the TRAC language came from the work of Eastwood and McIlroy [38] at Bell Laboratories. A system similar to the TRAC language which was developed independently in Great Britain is described by Strachey [39].

ununununununununununununununununununununununununununununununununununununununun

MODE OF OPERATION

There are two kind of functions: *primitives*, or machine-language subroutines that support the system in its environment. The primitives are the basis for the second type of function, called *forms*, or named procedures in Umist storage, which are character strings written like macro definitions and expanded, interpretively, when called. When writing a function call, one specifies whether its value (replacing the call) is to be processed again as part of the input string (*active* call), or whether processing is to continue starting with the portion of the string to the right of the value returned (*neutral* call). A single processing cycle is completed when the scanning and evaluating process reaches the right-hand end of the string.

Sequencing and evaluation in Umist are inherently recursive: function calls are evaluated from left to right, but may be nested to any depth in the arguments of other calls. Each function call is evaluated when, and only when, all of its arguments have been completely processed. Thus the string being processed is divided logically into two parts: the *active string*, consisting of input text (possibly preceded by inserted functional values) which is yet to be scanned, and *evaluated arguments of function calls* which are not completely ready for evaluation. This mode of operation, based on the completely interpretive execution of function calls, eliminates the distinction between program and data.

SYNTAX

Each function call in Umist has the form of a specially delimited argument list, in which the name of the function is always the first argument. Calls may be *open* (a variable number of arguments) or *closed*. A function call may be protected from evaluation by the use of literal delimiters. Another delimiter signals the right-hand end of the input string. These considerations lead to a syntax in which there are seven *special symbols*, whose occurances are deleted from the string during syntax scanning and whose presence indicates the beginning or end of a substring. The character strings enclosed in brackets below are the Umist special symbols:

1. Beginning of neutral function call [##(]
2. Beginning of active function call [#(]
3. End of argument [,]
4. End of call [)]
5. Beginning of literal [(]
6. End of literal [)]
7. End of input string [']

Note that the three beginning-of-substring symbols ##(and #(and (are terminated by the occurance of the same end-of-substring character,). Umist has a "parenthesis balanced" syntax, in the sense that an occurance of the right parenthesis matches only the *last* previous occurance of any one of the beginning-of-substring special symbols. Whenever a literal substring is encountered, the Umist processor removes the enclosing parentheses, but only the outer set is removed if more than one matching pair occurs. Thus a string initially protected from evaluation may be evaluated if scanned a second time, and, in general, evaluation can be controlled to occur the nth time the substring is scanned.

READ STRING AND PRINT STRING

The value of a 'read string' function call

```
#(RS)
```

is an input string accepted from the current input device. The 'print string' function

```
#(PS,X)
```

causes the display of the second argument, here symbolized by X, on the current output device, and has a null value.

When the Umist processor is first given control, and at the end of every processing cycle, the *idling procedure*

```
##(PS,#(RS))
```

is automatically loaded as an input string. This procedure first causes a read from the input device, with the input string becoming the second argument of the 'print string' call. Thus the string, if any, remaining when the input string has been completely processed, is finally printed before the idling procedure is again loaded. For example, if the input string is

```
#(PS,ABC)'
```

then after the 'read string' has been evaluated the processor is scanning the string

```
##(PS,#(PS,ABC))
```

and the inner call produces the output ABC ; the outer call nothing, since the inner 'print string' has a null value.

DEFINE, CALL, AND SEGEMENT STRING

Any character string in Umist can be given a name and placed in storage, from whence it can be called by using its name. The null-valued 'define string' function

#(DS,A,B)

places the string B in storage with the name A . A is called a *form* with value B . At most one string can be defined with a given name at any one time: use of the same name replaces a former definition. The value is retrieved with the 'call string' function

#(CL,A)

A form name, like a value, is any character string. The only restriction on length is that of the total string capacity of the processor.

The occurence of strings in storage is deleted with the 'delete definition' function

#(DD,N1,N2,...) .

This null-valued function removes the names N1, N2, ... as forms and discards their values.

Once defined, a form can be "parameterized," or *segmented*, using the 'segment string' function:

#(SS,A,X1,X2,...)

This null-valued function scans the form A , searching for an occurence of the string X1 as a substring. If X1 matches a part of A , that part is excluded from further matching, creating a "formal variable," or *segment gap*. The rest of the form is also compared with X1 to create, if possible, more segment gaps, all of which are assigned the *ordinal value* one, identifying the argument matched. The (separate) substrings of the form not already taken for segment gaps are next scanned with respect to the string X2 , and any occurences of the latter substring in A create segment gaps of ordinal value two, etc.

Thus, the 'define string' and 'segment string' functions together create a "macro" in which the segment gaps locate the "formal parameters." The "macro" is expanded by supplying the "actual parameters" in a call on the 'call string' function mentioned above:

#(CL,A,Y1,Y2,...)

The value of the 'call' is generated by returning the form A with all of the segment gaps of ordinal value 1, 2, ... replaced by Y1, Y2, ... respectively. If extra arguments are given in a CL, they are ignored. If some are missing, null strings are used as their values.

THE EQUAL FUNCTION

A decision function is provided for character strings:

#(EQ,A,B,T,F)

If the string A is identical to the string B , then the value of this function is the fourth argument, T; otherwise the value is the fifth argument, F. Since the strings T and F may be any Umist procedures, this primitive is the one normally used for branching.

APPENDIX B

TRAMP USER'S MANUAL

APPENDIX B: Table of Contents

APPENDIX B: TRAMP FUNCTIONS

This appendix is intended as a reference manual for Tramp and provides full specifications for using the various functions available in it. This section assumes familiarity with Umist (Appendix A), as well as an understanding of the design goals of Tramp as set forth in the body of this paper.

Running Tramp in MTS (Michigan Terminal System)

Tramp is inovked in the normal way by specifying it as the object file of a "RUN" command. The input is taken from the logical device SCARDS; output is put on the logical device SPRINT; and error comments (Tramp, not Umist) appear on the logical device SERCOM. While all three are global run parameters, the active input/output devices may be switched from SCARDS/SPRINT to some other logical device, dynamically within Umist, via the PAR function.

The "RUN" command can accept, besides these keyword parameters, a parameter list (via "PAR=") consisting of the following three global parameters, whose default values are underscored:

a. <u>NOPRIME</u> or PRIME

This parameter specifies whether or not the prime (') will be required to terminate Tramp input lines. If PAR=PRIME, then the program is in the normal Umist mode of operation: an input line is not terminated until the prime is encountered. Otherwise, a prime will automatically be appended to the end of each input line (if not already there), as delineated by a carriage return or other device-dependent end-of-record signal, by Tramp before it is passed on to Umist, which is still operating in the normal mode. If an input record has as its last character an ampersand (&), then that is taken to be a continuation mark: the ampersand is deleted from the line, which is passed on to Umist without a prime. If the ampersand is followed by a blank, then it is not a continuation mark; it must be the last character, not just the last non-blank character!

The mode of operation is initially set with this parameter, but may be dynamically altered during execution via the PRIME function, fully described in this appendix.

b. <u>*UMISTL</u> or *UMIST

This parameter specifies which version of Umist is to be used as the host interpreter. Presently the above two files are the two versions of Umist available. These two, and any other that might become available, may be used.

c. <u>NOW</u> or LATER

This parameter specifies when the Tramp functions are to be loaded. If PAR=LATER, then only Umist will be loaded initially, with the loading of Tramp being deferred until a call on #(tramp), further explained below.

Tramp in no way alters anything internal to Umist. With the exception of the function DSS (Define Special Symbol), the full facilities of Umist are available to the Tramp user. The Umist erasure functions, DA (Delete All) and RES (Restart), do not affect Tramp. If that is desired, you must also call ERM.

PRIMITIVE FUNCTIONS PROVIDED BY TRAMP:

NAME: DR

PROTOTYPE: #(DR,A,O,V)

PURPOSE: This is the associative storage function—the function that inserts the data into the structure.

DESCRIPTION: The three arguments, A, O, and V, are each non-empty sets. The set element delimiter in Tramp is the semicolong (;) because of the important role played by the comma in Umist. The triple is ordered and can be interpreted as meaning: A (O) = V. Each element of each set is grouped with each pair of elements of the other two sets, and the resulting triple is stored, i.e. each point in the cartesian product is stored. The three sets are ordered sets only inasmuch as the order in which they appear in the storage declaration is retained.

DR simply inserts the data into the structure in a way in which it can be efficiently retrieved. No check is made for inconsistency of data or for redundancies.

EXAMPLES: #(DR,AGE,MABLE;EUNICE,39)

this would store: AGE (MABLE) = 39

AGE (EUNICE) = 39

#(DR,COLOR,CAR,RED;GREEN)

NAME: KR

PROTOTYPE: #(KR,A,O,V)

PURPOSE: To undo what DR did—to erase an association from memory.

DESCRIPTION: The syntax of this function is exactly the same, and the effect exactly the opposite, of DR.

EXAMPLES: #(KR,AGE,#(RLR,AGE,**,*X*),##(X))

this would delete ALL associations containing "AGE" as the "A" component.

#(KR,COLOR,CAR,CHARTREUSE)

NAME: RL

PROTOTYPE: #(RL,A,O,V)

PURPOSE: This is the associative retrieval function. "Questions" are asked of the data structure by calling RL and specifying which, if any, among A, O and V are variables.

DESCRIPTION: Variables are denoted by enclosing a name, possibly null, within asterisks (*). To ask the question: "What color is the car?" one would write: #(RL,COLOR,CAR,**) or #(RL,COLOR,CAR,*NAME*) .

The "answer set" in this example is the set of all third componenets of associations having "COLOR" as the first component and "CAR" as the second. In the first instance above, the variable is not Named [nothing between the asterisks]. In this case, the answer set is the Value of the function. In the second instance, the variable is Named, which results in the function being Null-Valued, and the answer set being stored in Umist form storage labeled by the Name within the asterisks. Thus, the following two statements are exactly equivalent:

```
#(DS,ANS,#(RL,COLOR,CAR,**))
#(RL,COLOR,CAR,*ANS*)
```

If there are no variables, e.g. #(RL,COLOR,CAR,RED), then the question being asked is: "Does A(O) = V?", or in this case, "Is the car colored red?" No answer set is generated, rather a "truth value" is returned as the value of the function. If the specified association is in fact resident in the structure, or derivable thereof, then the value is "1"; if not, the value is "0." An ambiguity arises when one or more of the three sets has cardinality greater than one. Suppose #(DR,COLOR,CAR,RED) had been enterd. Then,

```
#(RL,COLOR,CAR,RED)        would have the value  "1"
#(RL,COLOR,CAR,BLUE)         "     "   "    "    "0"
#(RL,COLOR,CAR,RED;BLUE)     "     "   "    "    "?"
```

That is, the first association is found in storage, and the answer is "1." The second is not found, and the answer is "0." But two associations are specified by the last example, one is verified, the other not, and Tramp returns the value "?"

If there is one variable, then Tramp is being asked to "fill in the blank." The one variable may be in any of the three positions of the triple. The variable may be either named or unnamed, with the respective consequences described above.

If there are two variables, then two answer sets are generated. One of the variables is picked as the index variable, and values are one-by-one substituted for it, internally iterating on the one-variable question. The one constant may again be in any of the three positions of the triple. If both variables are named, the function is null-valued, and the two answer sets are stored and labeled by their respective names. If one is named and the other unnamed, then the set corresponding to the named variable is stored and the other answer set is the value of the function. It is syntactically valid for both variables to be unnamed, but this should not be done since then the value of the function would be the concatenation, not union, of the two answer sets.

The two-variable questions generate two answer sets—not a set of ordered pairs! Soon a variation of this function may be offered which will allow the generation of ordered pairs. In the meantime, if this is desired, the user will have to write a short Umist procedure to pick out the proper subset of the cartesian product of the two answer sets.

The present form of the two-variable questions—generating two answer sets—is very often used to find *all* "objects" associated with some other "objects," without regard for the third component of the triple. For example, to find the domain of the relation SON, i.e. all those who have sons, one could say:

```
#(RL,SON,**,*X*)
```

with the set of all sons now being stored in the form "X." In general, this generated set, here the set X, will not be further used, is not wanted, and wastes processing time. For this reason Tramp recognizes one special named variable for *two-variable questions;* "@," as denoting that the corresponding answer set is not to be generated. Thus,

#(RL,SON,**,*SONS*)	would return the set of all those who have sons, and store the set of all sons in the form "SONS"

```
#(RL,SON,**,*@*)    would likewise return the set of all
                    those who have sons, but would discard
                    the set of sons.
```

If there are three variables, it is interpreted as being a request for a dump of the associative memory. If any of the three variables are named, the names are ignored. Alternatively, one can simply call: #(DUMP).

```
EXAMPLES:    #(RL,*REL*,JOHN,HARVEY)  put the set of all relations that
                        associate John with Harvey in the form
                        "REL."

             #(RL,SON,CLYDE,**)   return the set of Clyde's sons.

             #(RL,COLOR,**,*COLOR*)  return the set of all objects that
                        have the attribute "COLOR," and place the
                        set of all colors in the string "COLOR."

             #(RL,*X*,*Y*,*Z*)  give a dump of the associative memory.
                        The three names are ignored.

             #(RL,AGE,#(RL,FA,#(RL,WIFE,#(RL,BRO,MARY,**),*WIVES*)
               ##(WIVES),**),**)'    recursively asks the question: "How
                        old are the fathers of the wives of Mary's
                        brothers?" Also, the set of wives of Mary's
                        brothers is now in the string "WIVES."

             #(RL,COLOR,**,*@*)   return the set of all objects that have
                        the attribute "COLOR," but do not generate
                        the set of colors.

             #(RL,COLOR,CAR,*@*)  put the set of the colors of the car in
                        the string "@."  "@" is a special symbol
                        only in the two-variable questions.
```

NAME: RLI

PROTOTYPE: #(RLI,A,O,V)

PURPOSE: To retrieve only implicit associations.

DESCRITPTION: This function has the exact same syntax as RL. It does not retrieve explicit associations. This is effected by deleting the "prefix" when expanding the program templates.

NAME: RLE

PROTOTYPE: #(RLE,A,O,V)

PURPOSE: To retrieve only explicit associations.

DESCRIPTION: This function has the exact same syntax as RL. It does not retrieve implicit associations. Calls to RLE bypass the preprocessor, thus ignoring any definition that may have been given for a relation. A normal call to RL calls both RLI and RLE.

NAME: RLR

PROTOTYPE: #(RLR,A,O,V)

PURPOSE: To retrieve answer sets that may contain redundancies

DESCRIPTION: This function is identical to RL except that any redundancies are reported. RL returns non-redundant answer sets, while RLR does not check for redundancies, and is therefore significantly faster.

NAME: INT

PROTOTYPE: #(INT,A,O,V)

PURPOSE: To generate intersections of answer sets.

DESCRIPTION: This function has the same syntax as the one-variable question of RL. RL generates the *union* of the answer sets, while INT generates the *intersection*:

#(RL,SOUTH;WEST,TOLEDO,**) generates the set of all things *either* south *or* west of Toledo.

#(INT,SOUTH;WEST,TOLEDO,**) generates the set of all things *both* south *and* west of Toledo.

If both constant sets are singletons, INT and RL will yield identical answer sets. The variable may again be in any of the three positions and may be either named or unnamed. This function must have *exactly one* variable.

EXAMPLES:

#(INT,NORTH;EAST,CHICAGO,*NE*)
place the set of everything both north and east of Chicago in the form "NE."

#(INT,**,JOHN;MARY,CLARA)
return the set of all relations that John and Mary commonly share with Clara.

NAME: RCOM

PROTOTYPE: #(RCOM,SET1,SET2,NAME)

PURPOSE: To compute the relative complement of two Tramp sets.

DESCRIPTION: The third argument is logically subtracted from the second argument, with the disposition of the resulting set determined by the fourth argument: if it is present, the function is null-valued and the set is stored in Umist form storage labeled by the name; if the fourth argument is omitted, the relative complement of the other two arguments is returned as the value of the function. The set computed consists of all elements of "SET1" that are not elements of "SET2."

EXAMPLE:

```
#(RCOM,#(RL,AGE,**,40),#(RL,SPOUSE,**,*@*),SPINSTER)
                          this would store in the form "SPINSTER"
                          all those who are 40 years old and not
                          married.
```

NAME: SYMD

PROTOTYPE: #(SYMD,SET1,SET2,NAME)

PURPOSE: To compute the symmetric difference of two sets.

DESCRIPTION: The symmetric difference of two sets is defined to be the set of all things that are in either of the two sets, but not in both (exclusive OR). The syntax of SYMD is identical to that of RCOM, with the fourth argument determining what will be done with the computed set.

EXAMPLE:

```
#(SYMD,#(RL,BRO,**,*@*),#(RL,SIS,**,*@*))
                 this would return the set of all those
                 who have siblings, but siblings of only
                 one sex.
```

NAME: INT

PROTOTYPE: #(INT,SET1,SET2,NAME)

PURPOSE: To intersect two Tramp sets.

DESCRIPTION: This function has the same syntax as the other two set operators, RCOM and SYMD. The two operands are the second and third arguments (SET1 and SET2) and the fourth argument specifies the disposition of the result: if it is present, it will be used as the name of the form into which the answer will be placed; if omitted, the answer will be returned as the value of the function. The answer is a straight set intersection, except that any redundancies are deleted.

Note that this function has the same name as the retrieval function INT. There is no ambiguity and there should be no confusion, since the two functions have dissimilar syntax. The retrieval function INT is called by specifying exactly three functional arguments, of which exactly one is a variable; the set operation INT is invoked by giving either two or three functional arguments, of which *exactly zero* are variables. I.e. a variable specifies retrieval—if there is no variable then a question is *not* being asked.

EXAMPLES:

```
#(INT,#(INT,NORTH;EAST,CHICAGO,**),#(INT,SOUTH;WEST,MAINE,**),UNHUH)
                              recursively uses both forms of INT to place
                              the set of all things both northeast of
                              Chicago and southwest of Maine in the form
                              "UNHUH."

#(INT,#(RL,AUTHOR,**,GEORGE),#(RL,SUBJECT,**,SEA))
                              this returns the set of everything that
                              George wrote about the sea.
```

NAME: DUMP

PROTOTYPE: #(DUMP) or #(RL,**,**,**)

PURPOSE: To obtain a complete listing of everything that is explicitly stored in the associative memory.

DESCRIPTION: All associations explicitly stored are printed out, using the "A (O) = V" format. A and O are singletons and V is the set of all "values" associated with the A/O pair. Any redundancies in the V set are printed. Implied associations are not listed in the dump. After all of the associations are listed, all of the current relational definitions are displayed.

EXAMPLES:

```
#(DUMP)
#(RL,*X*,**,**)
```

NAME: USE

PROTOTYPE: #(USE,NAME)

PURPOSE: To obtain the number of explicit associations that the Name is used in.

DESCRIPTION: The value of the function is the total number of associations that the name in the argument is used in. Any implied associations are not included in the USE count. There is no breakdown as to how the name is used within the associations, simply a count of the triples in which it appears.

EXAMPLES:

```
#(USE,COLOR)    how many objects have the attribute Color?
#(USE,JOHN)
```

NAME: ERM

PROTOTYPE: #(ERM)

PURPOSE: To completely erase the memory for a fresh restart.

DESCRIPTION: It is not anticipated that this function will be called very often, if ever, and to prevent its being invoked unintentionally, via misspelling, etc., confirmation is required by Tramp before it actually erases the structure. This is similar in form and in content to the confirmation that MTS requires before EMPTYing a file: an exclamation point (!) or the two letters "OK" are positive confirmation. Anything else cancels the request.

The above confirmation procedure is useful during debugging, but in a production program written in Tramp it will likely be desirable to provide your own confirmation procedure or eliminate it entirely. In such cases, ERM should be called with an argument. That argument is the single hexadecimal (non-graphic) character '3C'. This is easily done in Umist:

```
#(ERM,#(XTC,3C))
```

and is hardly likely to happen by accident.

NAME: CT

PROTOTYPE: #(CT,SET)

PURPOSE: To determine the cardinality of a Tramp set.

DESCRIPTION: This is a very simple function that is significantly faster and more convenient than a Umist procedure that would do the same thing. It distinguishes between: a missing argument; a null set; and a singleton (set with no semicolons); but otherwise is simply an efficient way to count semicolons.

EXAMPLE #(CT,#(RL,SON;DAUGHTER,SHERMAN,**)) How many children does Sherman have?

NAME: TABLE

PROTOTYPE: #(TABLE,X)

PURPOSE: To obtain the contents of one of the name tables.

DESCRIPTION: The argument, X, specifies which of the four name tables is desired:

A	- Attribute
O	- Object
V	- Value
D	- Defined relation

The set of names found on the particular table is returned as the value of the function.

EXAMPLES: #(RCOM,#(TABLE,A),#(TABLE,D)) return the set of all names that have been used as 'attributes' but have not been given definitions. The Defined relation name table is *always* a subset of the "A" name table.

#(INT,#(INT,#(TABLE,A),#(TABLE,O)),#(TABLE,V)) return the set of all things that have been used at some time in each of the three positions of the associative triple.

#(SYMD,#(TABLE,A),#(TABLE,V)) return the set of all names used as either 'attributes' or 'values' but not as both.

NAME: TRAMP

PROTOTYPE: #(TRAMP)

PURPOSE: To load Tramp if PAR specified that loading was to be delayed.

DESCRIPTION: If PAR=LATER, then only Umist will be loaded initially. When ready for Tramp, the user issues this function call which loads and links up all the Tramp functions.

The function TRAMP is defined only when PAR=LATER, and then only until it has been called.

NAME: PRIME

PROTOTYPE: #(PRIME,[ON,OFF])

PURPOSE: To set or invert the mode of operation regarding the prime that terminates all Umist input lines.

DESCRIPTION: An internal switch in Tramp determines whether or not a prime is required to terminate an input line. This switch is initially set by the parameter in the RUN command. The function PRIME may be used to alter dynamically the setting of this switch during execution. Normally, the switch should be ON while the program is being initialized (forms defined, etc.) and then is turned OFF just before reading the first input line from the user of the program.

The argument to PRIME may specify that this switch is to be turned ON or OFF, or simply inverted from its present setting. #(PRIME,ON) turns the switch ON, i.e. it specifies that a prime will be required to mark the end of a line. #(PRIME,OFF) sets the switch the other way, equivalent to: PAR=NOPRIME. Full details of operation with the switch off appear in the introduction to this appendix.

EXAMPLES:

```
#(PRIME,OFF)
#(PRIME)  no argument inverts the switch
#(PRIME,ON)
#(PRIME,X)  an unrecognizable argument inverts the switch.
```

NAME: SRCH

PROTOTYPE: #(SRCH,STRING,PATTERN,T,F)

PURPOSE: To provide pattern matching facilities along the lines of SNOBOL. There is no replacement (Umist segment string and call function provide replacement), but a string may be scanned for the occurence of a substring, and a decision made on that basis.

DESCRIPTION: The second argument, STRING, is scanned for an occurence of the third argument, PATTERN. If it is found the value of the function is the fourth argument, T, otherwise the fifth argument, F.

EXAMPLE:

```
        #(SRCH,##(FORM),?,(#(FUNC1)),(#(FUNC2)))
                           searches the string named FORM for an
                           occurence of a question mark.  If found
                           "branch" to the procedure FUNC1, other-
                           wise control goes to procedure FUNC2.
```

NAME: SAVE

PROTOTYPE: #(SAVE,FDNAME,RLNG,ID)

PURPOSE: To save the current state of the data structure on an auxiliary device so that at a later date the structure can be initialized to contain the present data.

DESCRIPTION: FDNAME is the name of the file or device onto which the data are to be SAVEd. RLNG is an optional argument specifying the record lengths to be written. If this argument is either omitted or specifies too large a record length for the particular device, the following default values, which are the respective physyical maximums, will be used:

PUNCH	80
FILE	255
TAPE	32,760

If RLNG = 80, either explicitly or by default, then each record will contain 72 bytes of information and 8 characters of sequential identification, the first 4 of which may optionally be specified in the last argument, ID. If more than 4 characters are given, extra characters on the right will be truncated. If less than 4, trailing blanks will be appended. If RLNG = 80 and this argument is omitted, the 4-character MTS signon ID will be used. If RLNG = λ, $\lambda \neq 80$, then there will be λ bytes of information with no identification.

EXAMPLES:

```
                #(SAVE,MYFILE)
                              write 255-byte records into the file.

                #(SAVE,MYFILE,80,IDX)
                              write 80-byte records into the file with
                              the specified ID.  Can now be copied to
                              a card punch.

                #(SAVE,*PUNCH*,,IDZ)
                              punch the data onto cards with "IDZ" ID.

                #(SAVE,*PDN1*,80)
                              write 80-byte records onto tape using
                              MTS signon ID.

                #(SAVE,*PDN2*,255)
                              write records on tape that can be copied
                              into a file.
```

NAME: COPY

PROTOTYPE: #(COPY,FDNAME)

PURPOSE: To read back in what has previously been SAVEd.

DESCRIPTION: COPYing in a new structure completely erases anything that might be in the structure at the time the COPY is called. There is no direct way to merge two Tramp data files. The following procedure is one way that two data files can be merged. Assume that DATA1 and DATA2 are the two files to be merged:

```
#(PRIME,OFF)'
#(COPY,DATA1)
#(PAR,FDO,SCRATCH)#(DUMP)#(PAR,FDO,*SINK*)
#(COPY,DATA2)
#(DS,PARSE,(#(DS,X,##(RS))#(EQ,##(CC,X), ,(#(SS,X, )&
     #(DR,##(CS,X),#(CS,X),#(NL,#(CS,X))##(CS,X))&
     #(PARSE)),(#(PAR,FDI,*SOURCE*)))))
#(PAR,FDI,SCRATCH((2)))#(PARSE)
```

A second routine, very similar to PARSE is then required to read in the dumped relational definitions.

NAME: PAGE

PROTOTYPE: #(PAGE)

PURPOSE: To ascertain what size file will be required to SAVE in and/or how much core the data structure is occupying.

DESCRIPTION: PAGE is a null-valued printing function which prints on the current output device. The output is the number of pages currently in core that will have to be saved, and how many extensions have been made to Tramp. The sizes of the various tables used by Tramp are assembly parameters and are likely to change. Presently the tables occupy a total of 4 pages of core. The information printed by PAGE is the amount of core being used in addition to the tables (tables cannot grow during execution).

Tramp is initially loaded with an Available Storage List 8 pages long (32,768 bytes). As this is used up, more is acquired from the system in blocks of 8 pages, called extensions. There can be up to 16 extensions (presently meaning that a maximum of 132 pages = 540,672 bytes would have to be SAVEd). These 8-page blocks are never broken up—SAVEing requires that the entire block(s) be written. In summary, (assuming 4 pages for tables) there is a minimum of 12 pages (= 49,152 bytes) and a maximum of 132 pages (= 540,672 bytes), with the minimum approaching the maximum in steps of 32,768 bytes.

NAME: DDR

PROTOTYPE: #(DDR,(REL = EXP))

PURPOSE: To define a relation in terms of other relations, thereby creating implicit associations in the associative structure.

DESCRIPTION: In the prototype, REL is the relation being defined, and EXP is an expression which is the definition. The equal sign is the delimiter and must be present. The interpretation of "REL = EXP" is that the ordered pairs specified by the expression EXP are taken to be a subset of the ordered pairs of REL. In the prototype the entire argument to DDR is enclosed in parentheses, i.e. a Umist "literal." Depending on the particular definition, this may or may not be necessary, but it will never hurt and it is good practice to always parenthesize the argument.

EXP is composed of one or more relations joined by the logical connectives: .A. (conjunction); .V. (disjunction); .N. (negation); two relational operators: / (slash meaning composition); .CON. (converse); and equality operators .EQ. and .NE. with obvious meanings.

The "R(x,y)" format is the relational format adopted by Tramp and is interpreted to mean that R (x) = y in the associative format.

The "converse" operator simply inverts the order of the two relational arguments: R(x,y) <==> .CON. R(y,x). Thus "child of" is the converse of "parent of," any symmetric relation is its own converse, etc.

The composition operator is defined by:

$$(x)(y)[(S/T)(x,y) \iff (\exists z)(S(x,z) \wedge T(z,y))]$$

DDR is the only Tramp function that allows spurious blanks. Before compiling the definition, all blanks are removed. In all other functions (except EDIT, below, which is another entry to DDR), blanks are valid EBCDIC characters and are treated like any other. Definitions may be either abbreviated (only the names of the relations specified—no relational arguments); or expanded (all of the relations given arguments in parentheses). There is the restriction that any one definition be consistent, e.g. #(DDR,(R1 = R2(X,Y)) is not legal. The same relation may be defined any number of times with each new definition being *appended* to the old, not *replacing* it, so that one can enter:
#(DDR,(R1 = R2 .V. R3))#(DDR,(R1(X,Y)=R4(Y,X)).

The two relational operators, composition and converse, may be used only in abbreviated definitions where there are no explicit relational arguments, since the purpose of these operators is to give information for expanding the relations to have arguments. On the other hand, the equality operators may only be used with the relational arguments as their operands, and hence may appear only in expanded definitions.. A constant which is to be used as a relational argument is denoted by enclosing the name of the constant in double quotes ("). The names of the dummy arguments used in expanded definitions may be up to 8 characters long.

Precedence of the operators: the precedence ordering of the various operators is as follows, descending order.

.CON.	converse
/	composition
.EQ., .NE.	equality
.N.	negation
.A.	conjunction
.V.	disjunction

The above precedence ordering may be altered in the usual way by the appropriate use of parentheses.

One may specify a "global" complement, e.g.

```
#(DDR,(R = .N. S)
```

which shall be ignored unless and until some other definition is given for R. (See discussion on page 135.) When multiple definitions are made for the same relation, the several expressions are ORed together—except in the case where one of the definitions specifies a global complement. In that case the global complement is ANDed with the others, if any, or else ignored if there are no others. If a new definition is entered for an already defined relation, any compilation error in the new definition results in a diagnostic being printed and retention of the old definition.

EXAMPLES:

```
#(DDR,(HUSBAND = .CON. WIFE))
#(DDR,BIGGER = LARGER)
#(DDR,(PARENT = FATHER .V. MOTHER))
#(DDR,(SPOUSE(X,Y) = CHILD(X,Z) .A. PARENT(Z,Y) .A. X.NE.Y))
```

for further examples, see Figure 4, page 62.

NAME: KDR

PROTOTYPE: #(KDR,REL1,REL2,REL3, ...)

PURPOSE: To erase definitions made by DDR.

DESCRIPTION: KDR may have any number of arguments. The definition for each of the relation names given as arguments is deleted.

EXAMPLES:

```
#(KDR,SIBLING)
#(DS,X,#(TABLE,D))#(SS,X,;)#(KDR,##(X,(,)))
```

would erase All definitions

NAME: SHOW

PROTOTYPE: #(SHOW, RELATION)

PURPOSE: To display the current definition of a relation.

DESCRIPTION: SHOW will display the definition of the relation specified by its argument exactly as it was entered by the user, except that blanks will have been removed. If more than one definition has been given for the relation, they will all be concatenated, separated by a break character, and displayed in a continuous line. Three actions may be taken by SHOW : if the relation has been successfully defined, its definition will be displayed on the current output device; if Tramp has never heard of the relation, the comment:

RELATION XXX <u>HAS NOT BEEN DEFINED</u>."

will be printed; if the relation was unsuccessfully defined, or was erased via KDR, the comment:

RELATION XXX <u>IS UNDEFINED</u>."

will be printed.

NAME: DDEF

PROTOTYPE: #(DDEF)

PURPOSE: To display all current relational definitions.

DESCRIPTION: DDEF iteratively calls on SHOW for each name found in the name table of defined relations. DDEF is an entry to the second half of DUMP which bypasses the listing of the associations.

NAME: EDIT

PROTOTYPE: #(EDIT,RELATION,PATTERN,REPLACEMENT)

PURPOSE: To correct or alter a relational definition made by DDR.

DESCRIPTION: The second argument, RELATION, is the name of the relation that is to be EDITed. The third argument is the pattern within the definition, as displayed by SHOW, that is to be altered. If this argument is null, it matches the void immediately to the right of the relation name in the definition string. The last argument is the string that replaces the pattern specified by the third argument. Any blanks in the PATTERN or REPLACEMENT will be ignored. If the last argument is omitted, the pattern is simply deleted. If the string specified by the third argument occurs more than once in the definition, only the first occurence is changed.

Calling EDIT implicitly calls SHOW to display the EDITed definition.

EXAMPLES:

```
#(EDIT,SIB,(.V.),(.A.))
```

change the first OR to AND in the definition of SIB. Like DDR, it is good practice to enclose the arguments in parentheses.

```
#(EDIT,REL,(.A. R4))
```

delete the string ".A.R4" from the definition of REL.

REFERENCES

1. Simmons, R.F., "Natural Language Question-Answering Systems: 1969," C. ACM 13-1, January 1970, pp. 15-30.

2. Green, C.C., "The Application of Theorem Proving to Question-Answering Systems," Technical Report No. CS 138, Stanford Artificial Intelligence Project, Stanford University, June 1969.

3. Cooper, W.S., "Fact Retrieval and Deductive Question-Answering Information Retrieval Systems," J. ACM 11-2, April 1964, pp. 117-137.

4. Kiviat, P.J., R. Villanueva, and H.M. Markowitz, "The SIMSCRIPT II Programming Language," The RAND Corp., R-460-PR, October 1968.

5. Levien, R.E. and M.E. Maron, "A Computer System for Inference Execution and Data Retrieval," The RAND Corp., RM-5085-PR, September 1966.

6. Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle," J. ACM 12-1, January 1965, pp. 23-41.

7. Ash, W.L. and E.H. Sibley, "TRAMP: An Interpretive Associative Processor with Deductive Capabilities," Proc. 23rd Nat'l Conf. ACM, Las Vegas, August 1968.

8. Elliott, R.W., "A Model for a Fact Retrieval System," University of Texas TNN-42, Austin, May 1965.

9. Brown, J.S., "A Symbiotic Theory Formation System," Ph.D. thesis, The University of Michigan, Ann Arbor, August 1971.

10. Simmons, R.F., "Answering English Questions by Computer," C. ACM 8-1, January 1965, pp. 53-70.

11. Levien, R.E. and M.E. Maron, "Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval," The RAND Corp., RM-4793-PR, December 1965.

12. Levien, R.E. and M.E. Maron, "A Computer System for Inference Execution and Data Retrieval," C. ACM 10-11, November 1967, pp. 715-721.

13. Maron, M.E., "Relational Data File I: Design Philosophy," The RAND Corp., P-3408, July 1966.

14. Ash, W.L., "A Compiler for an Associative Object Machine," Technical Report No. 17, Concomp Project, The University of Michigan, May 1969.

15. Mooers, C.N., "TRAC, a Procedure-Describing Language for the Reactive Typewriter," C. ACM 9-3, March 1966, pp. 215-219.

16. Weizenbaum, J., "Symmetric List Processor," C. ACM 6-9, September 1963, pp. 524-536.

17. Kassler, M., Review No. 18,534 (C.C. Green), *Computing Reviews* 11-2, February 1970, pp. 112-113.

18. Feldman, J.A., "Aspects of Associative Processing," Technical Note 1965-13, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, April 1965.

19. Feldman, J.A. and P.D. Rovner, "An ALGOL-Based Associative Language," C. ACM 12-8, August 1969, pp. 439-449.

20. Swinehart, D. and B. Sproull, "SAIL," Stanford Artificial Intelligence Project, Operating Note No. 57, November 1969.

21. Morris, R., "Scatter Storage Techniques," C. ACM 11-1, January 1968, pp. 38-44.

22. Peterson, W.W., "Addressing for Random-Access Storage," *IBM Journal of Research & Development*, April 1957, pp. 130-146.

23. *The University of Michigan Terminal System Manual*, 2nd edition, Vols. I & II, The University of Michigan Computing Center, Ann Arbor, December 1967.

24. Ash, W.L., "An OS/360 version of TRAMP," Internal Memo, Concomp Project, The University of Michigan, January 1969.

25. Rovner, P.D., "An Investigation into Paging a Software-Simulated Associative Memory System," Sc.M. thesis, The University of California, Berkeley, 1966.

26. Feldman, J.A. and D. Gries, "Translator Writing Systems," C. ACM 11-2, February 1968, pp. 77-113.

27. Floyd, R.W., "Syntactic Analysis and Operator Precedence," J. ACM 10-3, July 1963, pp. 316-333.

28. Floyd, R.W. "Bounded Context Syntactic Analysis," C. ACM 7-2, February 1964, pp. 62-67.

29. Ford, L.R.Jr. and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, 1962.

30. Busacker, R.G. and T.L. Saaty, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, 1965.

31. Allan, J.J. "Man-Computer Synergism for Decision Making in the System Design Process," Ph.D. dissertation, The University of Michigan, Ann Arbor, June 1968.

32. Sibley, E.H., R.W. Taylor and D.G. Gordon, "Graphical Systems Communication: An Associative Memory Approach," AFIPS FJCC, 1968, pp. 545-555.

33. Levien, R.E., "Relational Data File: Experience With A System for Propositional Data Storage and Inference Execution," The RAND Corp., RM-5947-PR, April 1969.

34. Turing, A.M., "Can A Machine Think?" reprinted in Feigenbaum and Feldman (eds.), *Computers and Thought*, McGraw-Hill, New York, 1963.

35. Slagle, J.R., "Experiments with a Deductive Question-Answering Program," C. ACM 8-12, December 1965, pp. 792-798.

36. Raphael, B., "SIR, A Computer Program for Semantic Information Retrieval," in Minsky (ed), *Semantic Information Processing*, MIT Press, Cambridge, 1968.

37. Mooers, C.N. and L.P. Deutsch, "TRAC, A Text Handling Language," Proc. ACM Nat'l Conf., Cleveland, August 1965, pp. 229-246.

38. Eastwood, D.E. and M.D. McIlroy, "Macro Compiler Modification of SAP," Computer Lab. Memo, Bell Telephone Labs., Murray Hill, N.J., September 1959.

39. Strachey, C., "A General Purpose Macrogenerator," *Computer Journal* 8-3, 1966.

40. Coles, L.S., "Talking with a Robot in English," Proc. *International Joint Conference on Artificial Intelligence*, Washington, D.C., May 1969.