GENERALIZED ORGANIZATION OF LARGE DATA-BASES; A SET-THEORETIC APPROACH TO RELATIONS

Andrew Irwin Fillat

and

Leslie Alan Kraning

June, 1970

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

GENERALIZED ORGANIZATION OF LARGE DATA-BASES; A SET-THEORETIC APPROACH TO RELATIONS*

Abstract

Problems inherent in representation and manipulation of large data-bases are discussed. Data management is considered as the manipulation of relationships among elements of a data-base. A detailed analogy introduces concepts embodied in a data management system. Set theory is used to describe a model for data-bases, and operations suitable for manipulation of relations are defined. The architecture chosen for an implementation of the model is illustrated, and a representation of data-bases is suggested. A particular implementation, the GOLD STAR system, is investigated and evaluated. The framework outlined is meant to provide an environment in which complex data handling problems can be solved with relative ease. GOLD STAR provides the user with tools sufficient for manipulation of arbitrarily complex data-bases; these provisions are presented in the form of an extremely simple interface.

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

ACKNOWLEDGEMENT*

The authors wish to thank several individuals whose interest and cooperation made the thesis possible:

Our energetic advisor, Professor James D. Bruce, for his belief in the merits of our research, and for his continuing faith in us.

Our mentor and colleague, Burton J. Smith, for his guidance, helpful suggestions, and tutelage.

Our typist, Nancy J. Murphy, for her adroit translation of the manuscript into English.

Each other, for faithful interlocution and cooperation throughout the research.

*This work was performed at Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, and was supported in part by the Office of Naval Research under Contract N00014-69-0276-0002.

TABLE OF CONTENTS

LIST OF FIGURES

This empty page was substituted for a blank page in the original document.

CHAPTER I

INTRODUCTION

The M.I.T. Electrical Engineering Department, like other large organizations, requires a powerful data management capability to provide information for daily operation. The department currently utilizes a software system developed by Burton J. Smith[1-1] which operates on the M.I.T. Compatible Time-Sharing System. Faced with the burden of manual generation of teaching and research assignments, budget reports, mailing labels, and other documents, the department began in 1966 to transfer these tasks to the CTSS system with a significant saving in production time, personnel effort, and cost. With the forthcoming demise of the 7094 CTSS installation, an alternative computer system is needed, and provides the motivation for this thesis.

For any integrated information system, the flexibility, and hence the scope of its utility, depends heavily upon the sophistication of its associated data management system. While this term connotes the existence of various functions to different individuals, the prime purpose of such an entity is to provide a coherent and systematic method of manipulating large volumes of interrelated data. Such a purpose is laudable, but only if a user can avail himself of the capability in a manner which makes the results, as well as the process, meaningful to him.

Our concern in this thesis is the development of a data management facility capable of serving a wide variety of administrative information needs. Specifically the system characteristics are:

1. Pre-eminence of User Perspectives and Techniques.

   A data management system is most effective when a user can express his problem to the system in terms of his perspective and biases; to the faculty planner, data types such as "name", "salary", and "project" hold much more meaning than representation types like INTEGER, DECIMAL, or BIT. For the same user, the operation "SORT (PHONE_BOOK, (TEL, NAME, ADDR))" expresses his purpose more than the effort required to write his own sorting routine. A good data management system will minimize the degree to which a user alters his perception of, and approach to, his information problems.

2. Modular Construction.

   While extensive planning of a system is advisable before development, experience indicates that software systems evolve with time rather than merely exist. That is, pre-planning a system rarely encompasses all features that may be desired at some unknown point in the future. A system should be modular in the sense that features added or

deleted in the evolution process disturb only local portions of the system. Modularity obviates the need for redesign and re-programming at each change.

3. Exportability

An organization may change the computer system it utilizes for data processing, but such a change should not disrupt the operation of the firm or division. While few computer programs are machine independent, a data management system can attain a high degree of exportability or relative machine independence, by isolating particular machine dependent features in a few modules. The remaining system modules should be written using a widely available higher level language. Design of a data management system with this objective in mind minimizes the operational disruption occasioned by a change in the target computer.

The goals outlined thus far apply to design philosophy; as such they pertain to all features which a data management system may exhibit. There are, however, more specific requirements placed upon any particular data management system, requirements dictated by the characteristic environment in which the system must operate. The envisioned system is best suited to a "computer utility"[1-2] environment, and as such requires two further considerations:

4. Controlled Sharing Among Users

A computer utility provides the services of computation for a broad-based community of users. Since some information, such as salaries and grades, is often considered personally sensitive, it is of prime importance that only those members of the community who require this information be allowed access to it. The strategy of allowing a user to access only these relations he creates would largely protect the integrity of sensitive information. However, such an extreme requirement eliminates effective communication among users whose data purviews overlap.

5. On-Line Capabilities

Management of a department or office occasionally requires instantaneous access to the data-base. The ability to update files from a remote terminal is extremely convenient; data may be entered by a secretary as if she were typing a report, obviating the need for keypunching and physical handling of cards. However, the greatest need for on-line access arises when a user searches the data-base for specific facts, and based upon the computer's response, asks for any of a wide range of additional data. For example, the university bursar may ask

for student accounts showing a lapse of payment greater than 30 days, along with the total number of dollars receivable from students. If the total due were mostly a result of the overdue accounts, he could generate address labels for those students tardy in payment and send notices using these labels. Interaction of a user with his data-base in this fashion is a powerful administrative tool, and thus, a justification for on-line capabilities.

The concurrent requirements of sharing and pre-eminence of user perceptions of relatedness present the sub-system designer with a formidable task. Vendors of software packages offer a variety of solutions but almost all efforts focus on representation and manipulation of data elements. GOLD STAR (Generalized Organization of Large Data-bases; a Set Theoretic Approach to Relations), however, focuses upon the representation and manipulation of relations among data elements. This focus should be the prime function of any data management system. The functions of the GOLD STAR data management system are: 1) to implement an arbitrary conceptualization of relatedness among entries in a data base, and 2) to transform the specified operations on the data base into sequences of instructions which will produce the effect of these operations. Thus the design intention of GOLD STAR is to remove, as far as possible, computer constraints on the conceptualization of a data management

problem, while providing a direct means of mechanizing the same.

Several methods of implementing such a system are possible; in the first effort, GOLD STAR is imbedded within the PL/1 language as a set of function sub-programs. Imbedding allows the user full access to GOLD STAR without the burden of mastering some esoteric programming language; rather it extends the power of an established programming system to the realm of relation management. Imbedding permits implementation of a model at low cost; the model's utility and power can be evaluated, as well as operated, without development of a special purpose compiler or interpreter. GOLD STAR assumes that calls to the imbedded functions are "primitives", i.e., a user need endure no greater complexity than these calls. GOLD STAR is written entirely in the PL/1 programming language and its facilities are available to any program written in PL/1.

CHAPTER II

THE STRUCTURE OF GOLD STAR

To explain the data-management system developed in this thesis, we shall use an example designed to illustrate and motivate the system structure we have chosen. By introducing additional concepts to the analogy, it will be possible to explain all of the major issues of the GOLD STAR system. Following the example, we will relate the user's needs to the data-management system by both describing the general kinds of operations he might wish to perform, and the way in which the system conforms to the goals outlined in the first chapter.

We assume that a small company has decided that it should establish a name-address-telephone number directory of its employees, although it has only eight employees. The preliminary, hand-written draft of the directory appears in Figure 2-1on the next page.

This preliminary draft of the directory has some very important properties which should be noted. First, each line not only contains three items of information (i.e., a name, an address, and a telephone number), but it states by its physical structure that each of these items is in a sense related, i.e., they "belong" to each other. Second, each item in the directory is associated, by its column, with a heading or type. Hence, "541-6622" (a telephone

Name	Address	Telephone
Abernathy, Fred	22 Maple Street	783-3055
Barnes, John	13011 S. Weymouth Drive	249-8112
Donnelly, Bill	2 Wallingham Place	247-7731
Jones, Art	53 Main Street	724-3718
Jones, Sally	53 Main Street	724-3718
Manning, Pete	264 Carling Avenue	861-8366
Manning, Pete	264 Carling Avenue	861-4431
Parker, Irv	10 S. Pannert	541-6622
Sanders, Doug	3 Mangrove Plaza	443-8190

Figure 2-1: Preliminary Draft of Directory

number) would never be found in the same column as "Parker, Irv" (a name). Third, the directory itself is characterized by the types it contains (in this case name-address-telephone). Finally, we note that the first column is in alphabetical order. This makes a line easy to find when given a name. Hence, "Jones, Sally" is much easier to locate than is "861-8366", since the directory is ordered primarily by names rather than by telephone numbers. The left-to-right representation of the directory is usually used to imply that the further left we look, the more order a column possesses. This is a convention adopted in most cases which is not innately necessary.

Now let us complicate the problem -- a printing strike is in progress when the directory is to be published. Printing costs are very high, with charges figured on a per character basis. In assessing the alternatives, management realizes that the company currently possesses a microfilm storage system for which each employee has a small console unit. These consoles each have an array of push-buttons which are used to display selected information on a screen.

Management decides upon the following approach to make the directory available to the employees: the names, addresses, and telephone numbers are placed on microfilm, and a list of which button is to be pressed on the consoles to retrieve information about which employee is readied for

printing. Thus, if you were to press the button "name" (this loads the "name" cartridge), and then the button "1", "Abernathy, Fred" would appear on the screen. In Figure 2-2 we picture the information stored on microfilm as well as that printed.

At this point the use of this directory is not necessarily clear. However, before considering how, we should observe a number of important concepts embodied by this change, and which motivate the change.

1. The headings in the printed directory (Figure 2-2-d) are still necessary in order to know which cartridge to load when information is requested. Thus an item of information is associated with both a button number and a (cartridge) type.

2. The only association between these cartridges and this directory are through the headers (and corresponding cartridge types). The same "name" cartridge would be used as part of this company's name-department listing. Note that a saving is reaped for this second listing in that a new "name" cartridge would be unnecessary.

3. Although one forms a mental picture of the contents of a cartridge such as is illustrated in the figure, we do not actually know how the information is physically stored. The extent

Cartridge No. 1: Loaded by pressing "name"

Button No. For Retrieval	Information Retrieved
1	Abernathy, Fred
2	Barnes, John
3	Donnelly, Bill
4	Jones, Art
5	Jones, Sally
6	Manning, Pete
7	Parker, Irv
8	Sanders, Doug

Figure 2-2-a: Name Console Cartridge

Cartridge No. 2: Loaded by pressing "address"

Button No. For Retrieval	Information Retrieved
1	22 Maple Street
2	13011 S. Weymouth Dr.
3	2 Wallingham Place
4	53 Main Street
5	264 Carling Avenue
6	10 S. Pannert
7	3 Mangrove Plaza

Figure 2-2-b: Address Console Cartridge

Cartridge No. 3: Loaded by pressing "telephone number"

Button No. For Retrieval	Information Retrieved
1	783-3055
2	249-8112
3	247-7731
4	724-3718
5	861-8366
6	861-4431
7	541-6622
8	443-8190

Figure 2-2-c: Telephone Console Cartridge

The Printed Directory

Name	Address	Telephone Number
1	1	1
2	2	2
3	3	3
4	4	4
5	4	4
6	5	5
6	5	6
7	6	7
8	7	8

Figure 2-2-d: Printed Directory

of our knowledge is that we can perform two functions: we can press a numbered button (presumably we have already loaded the cartridge) and have the corresponding data item appear on our screen, or we can somehow type in a data item (the hardware is irrelevant) and have its button number appear on the screen. There is no reason to believe that names and addresses are stored in the same physical manner in the cartridges, despite the fact that the console performs identically for all corresponding requests.

4. Some tasks become easier. In glancing over the directory, "Jones, Sally" will be more easily mistaken for "Jones, Art" than "5" will be for "4". In addition, comparisons are easier by mere virtue of the fact that directory entries are now a single number rather than a multi-character name (or address, etc.).

5. There is an implied order on button numbers (numeric, of course). Hence, in the case of "name" numeric order on the button corresponds to the alphabetic order on the names. However, in the case of "address", where no order was assumed in the beginning, the task of determining the button number associated with a

given data item is a hard one -- all address entries must be checked (i.e., there is no guarantee of any relative positioning for the addresses).

6. In cases where duplication of items occurs, large savings accrue from the given organization. For example, although Pete Manning has two lines in the directory, all that must be printed twice is his button number. His name is stored only once on the "name" cartridge.

7. The directory is meaningless out of context (i.e., with no headers). One would not know which cartridge to load before pressing the buttons for inquiry.

In order to consider how the system is used, we take a name and request the associated address and telephone number. First, the "name" button is pressed to load that cartridge. Then the desired name is inserted and the associated button number is read off the screen. Now we look in the printed directory for the line with that number in the name column and retrieve the other information by pressing the appropriate cartridge load button and number button.

Note that we have said that the printed directory in this numeric form retains exactly the same properties as did the original listing with the complete text (Figure 2-1).

There are new considerations (as outlined above), but there are no conceptual differences in the information content of the two methods of representation.

Let us now assume that negotiations in the printer's dispute have broken down altogether; now no printing services are available. Management can either wait for the printer's strike to end or seek an alternative. They decide on the following alternative: a new set of microfilm equipment is purchased, with each employee receiving a terminal (we shall distinguish these "terminals" from the previously discussed "consoles"). This new system (the terminal system) is to be employed to store the directory itself in a directory-cartridge, which will be named, appropriately, "name-address-telephone number". The facilities of the terminals must be more complex than the consoles in order to handle these more complex directory-cartridges.

The terminals are capable of responding to requests such as "display the line in name-address-telephone number containing "5" in the address column." The request itself is not necessarily in English, but the key words corresponding to load directory (first "in"), locate line ("display"), look in a given column (second "in"), look in a given column for a given button number ("containing") correspond to buttons. In addition, the terminals are equipped with the ability to read and create magnetically inked listings of the directory car-

tridges. These listings are used to verify the entire contents of a directory cartridge, or to insert a new cartridge, or to make modifications, or other such operations. Note that the listings produced and read are in exactly the same format as the one in Figure 2-2-d.

Now, again, we focus on some key considerations generated by the change in our information storage arrangement.

1. We still have no idea how a directory is physically stored on a directory-cartridge. We know only that the terminal can perform various tasks (which we will elaborate on later) on the directory-cartridges such as load them, inquire of them, and make a listing from them. The data items stored in these directory cartridges are the same button numbers which would have appeared in a printed directory (Figure 2-2-d) and are to be used on the consoles to obtain text from the screen. Further, there is no reason to assume that all directory cartridges have the same physical format.

2. The logical structure used in different directory cartridges may be different. The "relatedness" implied by the line structure may be recorded in ways other than lines. The only requirement to be met by the other logical structures is

that they be capable of being translated into a listing of the form of Figure 2-2-d. For example, Pete Manning's two different telephone numbers may both be associated with his name and address in such a way that the name and address columns need be physically present only once for his name.

3. More than one directory-cartridge can be used in the terminal system. One such cartridge has no relationship to another. It is possible to have both "name-address-telephone number" and "name-department" cartridges, but the existence of a given name in one says nothing about the existence of that name in the other (i.e., because a name possesses an address and number, we can determine no association between this name and any department).

Now consider the kind of operations we might want to perform upon the directory-cartridge(s). If another company merged with this one and it used the same information system, we might want to merge the two "name-address-telephone number" directory cartridges to form one. Or, if an employee left, we might wish to remove all lines about him from a directory-cartridge. Or, we might wish to "combine" the "name-address-telephone number" with "name-department" to give a new "name-

department-address-telephone number" directory-cartridge. These are but a few of possible operations which we might want to perform.

It would be premature at this time to discuss the full spectrum of operations we may wish to perform on the terminals and consoles. Rather, we extend our terminology by putting forth a general system structure for GOLD STAR and showing how the various components of the system correspond to our example. Then we may attempt to define at least the classes of operations we might do with the consoles and the terminals. The diagram of Figure 2-3, although it may seem cryptic at first, serves to characterize the GOLD STAR data management system.

Now let us take the various blocks of the figure and discuss their function by referring to the example of the microfilm system.

DSM (Data Strategy Module): These correspond to the consoles and serve to translate between the reference numbers (the button numbers) and the data items.

DA (Data Area): These correspond to the data cartridges which are part of the console system. Just as a button must be pressed to load a cartridge, so must information be given to the DSM to tell it which DA to refer to when retrieving text or reference numbers (refnos). This "load button" is what is

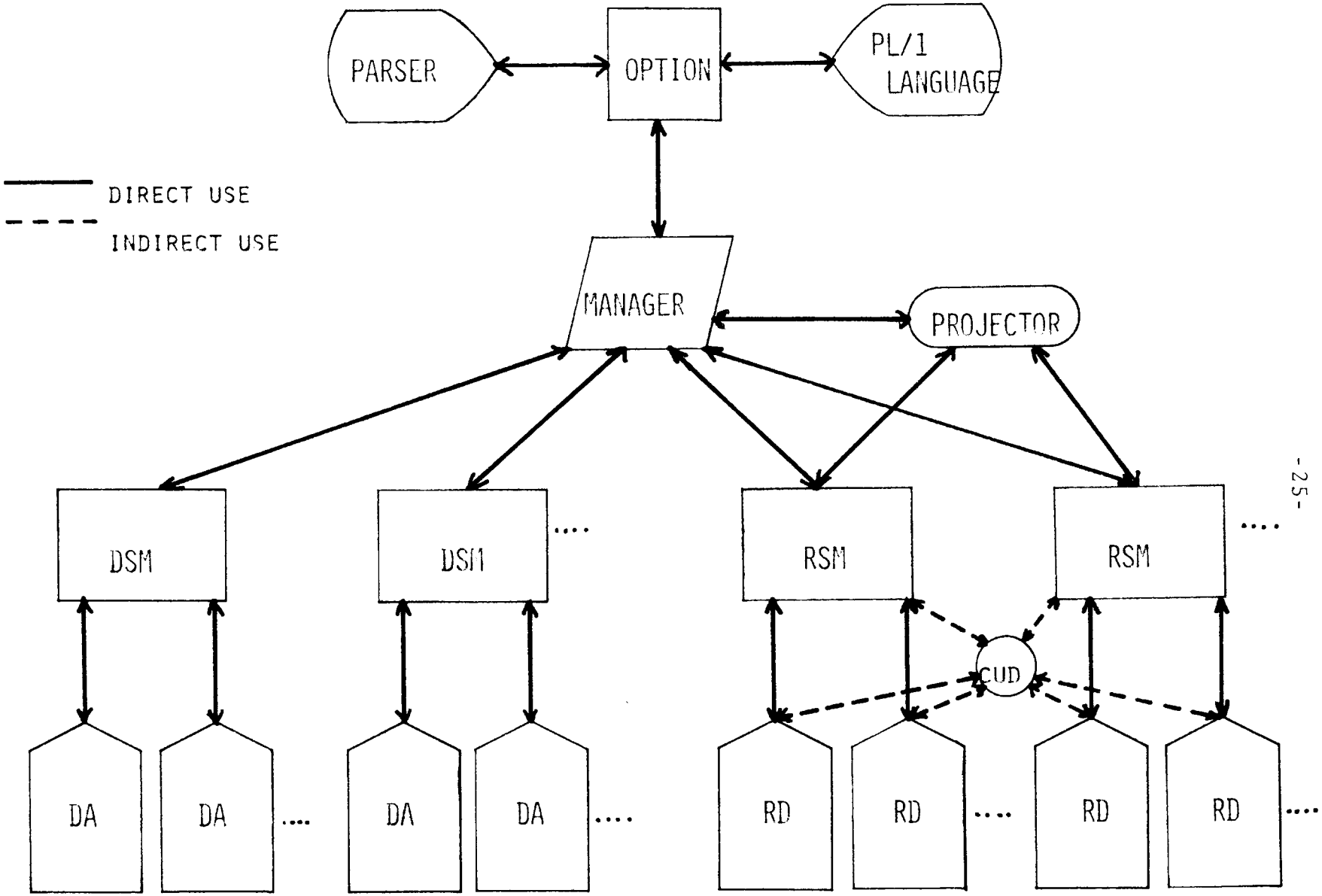


FIGURE 2-3 GOLD STAR SYSTEM STRUCTURE

referred to as a data-type name.

RSM (Relational Strategy Module): These correspond to the terminals and serve to perform operations on relational data just as the terminals perform operations upon the directory-cartridges.

RD (Relational Data): These are the directory-cartridges of GOLD STAR. Just as the directory-cartridge must be requested by name, so must an RD be requested via a relation name (or rel-name).

QUART (QUAsi RelaTion): These are the listings of the terminals. All RSM's can produce quarts or can, in fact, convert the contents of an RD to a quart. These are the fundamental units of the system in that all modules of the system can deal in some way with quarts.

MGR (Manager): There is no direct analogy to the manager in the microfilm system. The manager is merely the module which has the effect of making button-pressing operations easier. It provides an easy language for the user to use in utilizing the available powers of the DSM's and RSM's.

CENT & PENT: Two representations (character and pointer) of a header. The rest of the components serve much more specific tasks and will be discussed

in later sections. To briefly review the main concepts again, information text is stored only once and a token refno assigned to it. The information of how data items are related to one another is stored separately and is operated upon separately. This approach allows the savings and features mentioned in the discussion of the example to be capitalized upon.

Consider briefly the classes of operations which will be desired from our components.

$$\{\text{Data-String}\} \xrightarrow[\text{DSM}]{} \{\text{refno}\} \qquad (2\text{-}1)$$

This is the translation of text into its reference number (refno). We have assumed that the nature of the data string is known so that we can determine into which data area we must inquire. We have also assumed that each data area has one and only one DSM which can perform the indicated translations and that we can easily determine which DSM by looking in the data area.

$$\{\text{refno}\} \xrightarrow[\text{DSM}]{} \{\text{Data-String}\} \qquad (2\text{-}2)$$

This is the reverse of (2-1). The same assumptions hold true. It is worthwhile to note at this point

that each reference number is stored as part of a quart; these quarts also contain information indicating the data area to which this refno refers. Thus, the specification of data area can be either implicit (found from the refno's quart) or explicit.

$$\{\text{refno}\} \xrightarrow[\text{DSM}]{} \{\text{refno}\} \qquad (2\text{-}3)$$

This is the way of representing a transformation through a DSM, such as "get the next refno after the one I supply". Again, the same assumptions as above hold, and data area can be implicity or explicitly indicated.

$$\{\text{refno}\} \xrightarrow[\text{RSM}]{} \{\text{refno}\} \qquad (2\text{-}4)$$

This indicates a transformation performed when information (in the form of refnos) is given to an RSM; what is returned is another bundle of information which has been found using the properties of relatedness stored in a relation (relational data area). The appropriate RD and RSM are implicitly determined from a group of input refno quarts.

$$\{\text{refno}\} \xrightarrow[\text{RSM}]{} R \qquad (2\text{-}5)$$

This is the same as (2-4) except that the result is a complete RD containing the properties of the transformed refno input.

$$R \xrightarrow[RSM]{} \{refno\} \qquad (2\text{-}6)$$

This is the reverse of (2-5).

$$R \xrightarrow[RSM]{} R \qquad (2\text{-}7)$$

This is the transformation of one RD to another. A merge is an example of such an operation.

These are the general classes of operations available to the user. The first three are those which would be performed on the console in the example. The final four would be performed on a terminal where {refno} would be a listing and R would be a directory-cartridge. These operation classes are quite general, and later sections will narrow their scope to certain well defined procedures.

But now, how can we justify this rather complex structuring of the system in terms of our goals? Let us consider the goals one by one:

1. User Perspective

   The concept of data-types, or headers, identify each reference number as being associated with a given type of data. Hence, the user thinks of

his data in terms of their type, or header, rather than in terms of its PL/1 representation. This is exemplified by the fact that no matter how data is stored on a console cartridge, the user can handle it merely by pressing the button corresponding to the cartridge name; i.e., the machine handles representational considerations, and the user is free to imagine all his different kinds of data in terms of their type name.

2. Modularity

   The freedom to use virtually any organization of data or relations is guaranteed by this structure. All that is needed is an appropriate DSM or RSM to handle that format. The interfaces in the system are uniform and well defined; addition of new modules becomes little more than writing them. This fact is not easily accounted for in our model; we are proposing that new consoles and terminals are easy to add when new format cartridges are to be handled.

3. Exportability

   Of course, the major feature of the system for exportability is the PL/1 language (see Chapter I). Beyond that, two important facts remain. First, the manager can be designed to incorporate all of

the machine dependent code, leaving the RSM's, DSM's and information free for transplantation to any machine. Second, the interfaces are so explicit that any additional modules needed by a new machine could easily be interposed in a calling sequence between two system modules. There is no analogy to the example, except if the entire system were adopted by another company with different microfilm equipment.

4. Controlled Sharing

Clearly this system allows for sharing by mere sharing of data areas (cartridges). The control arises from the fact that access to relations can be given selectively. For example, an employee might have access to both the name and salary cartridge (Data Areas) but by being denied access to the name-salary relation cartridge (relational data) he has no way of making the associations. The advantage to this scheme is that the employee could well be permitted to examine other relations containing name and salary (e.g. name-address or salary-contract) and still never be able to associate the two. On the other hand, circumstances might dictate that a console cartridge (or data area) be denied a user, and still allow to him

all relations involving it. For example, we might not care if it is known from name-classification that ten people are of the same rank, but we might want to protect just what that rank is. In short, with all data and relational data stored in modular units, we need worry in general only about sharing these units, rather than about sharing parts of a larger combined medium.

5. On-Line Capabilities

   This is a function of how the system's functions are adapted for use, and does not bear on the system design itself.

As to the unstated goal of efficiency, there are a couple of points to consider. Appropriate design of RSM's and DSM's can optimize operation to conform to the most frequent function that the module must perform. For example, an RSM can be designed to optimize inversion, just as one can be designed to optimize retrieval operations of a uni-directional nature (i.e., with name-address-telephone, we are always given a name and asked for the other two, but rarely in any other order). Also, we achieve some saving of space by using reference number tokens in relations; duplication costs only the length of a refno, not of the data itself.

That summarizes the structures and motivations of the

GOLD STAR system. In the next chapter, we will consider the set theory upon which these structures are founded, and how set theory leads in logical progression to the concept of a quart.

Chapter IV will discuss the implementation of GOLD STAR and provide an insight into some of its current design capabilities. Chapter V will present the user interface, i.e., the functional calls to the manager of the system. That section also discusses the programming considerations of note, as well as presenting an example of the use of GOLD STAR.

CHAPTER III

SET THEORY OF GOLD STAR

1. The Rhetoric of Sets

Each individual perceives the "relatedness" of two items of data based upon his particular perceptions and biases. Visual proximity, concurrent sensations of taste and smell, or remembered patterns of association are but a few means by which the human mind establishes relatedness of two items of information. Different individuals often interpret the same collection of information in vastly different terms of relatedness; the string "V-8" as construed by the auto enthusiast signifies something afar from the meaning assumed by the connoisseur of vegetable juices.

At the core of GOLD STAR rests the firm belief that data relatedness is best conceptualized apart from the computational processes which implement relatedness. One conceptualization method, set theory, permits this separation, and offers several unexpected benefits:

-- Set theory is sufficiently general to encompass the operations and associations normally subsumed in the term "data management".

-- As an outgrowth of pure mathematics, set theory offers unambiguous, albeit complex, semantics; furthermore, set operations and their limitations are well established.

-- The subdivision of a system into independent functional modules is facilitated by a set-theoretic framework. Set-theoretic operations specify quite well the tasks that any single system module must perform.

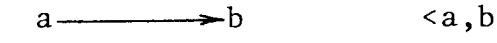
The remainder of this section discusses data-bases and operations upon them in terms of basic set theory. The following section treats data-bases in light of orderings upon their members.

Data-Bases

The notations and intuitive notions used in this section conform with common usage.[3-1] As an abstract model of reality, a data-base depends upon certain concepts involving sets, most importantly:

The set {a,b} is a collection of the objects "a" and "b" with neither ordering nor structure presumed among them. The elements "a" and "b" may be either atomic, i.e. not sets, or they may themselves name sets. Thus an element of a set can be another set.

The ordered pair <a,b> is the set {{a},{a,b}}, and indicates in the following directed graph that the correct sequence of endpoints is "a" followed by "b".



The tuple (a,b) is the set $\{<1,a>,<2,b>\}$, that is a function from {1,2} to {a,b}. Likewise, the tuple $(n_1,n_2,n_3,....n_i)$ is a function from the set $\{1,2,3,4,...i-1,i\}$ to $\{n_1,n_2,...n_i\}$.

We define a data-base B as a tuple of order three:

$$B=(D,T,R) \tag{3-1}$$

The set D contains data element names, and contains within its power set three particularly useful sets.

$$D=N_D \cup N_T \cup N_R \tag{3-2}$$

The members of N_D are raw data elements, that is elements apart from their relation to other elements. For example, the elements "John", "red", "3.5", and "$-2+3.83\sqrt{-1}$" could all be members of N_D.

The members of N_D, and particularly associations among its members, are the motivation for the structure of the data-base B. Of first concern in this structure is the function T, the data type function. T takes as its domain a set of data type names, N_T, $N_T \subseteq D$, and maps it into various subsets of D. More rigorously:

$$T=N_T \rightarrow \wp(D) \tag{3-3}$$

From the example of Chapter II,

N_T={"name","address","telephone"}

T(name)={"Abernathy, Fred","Barnes, John", etc.}

T(address)={"22 Maple St.","53 Main St.", etc.}

T(telephone)={"783-3055","724-3718", etc.}

T(data types)={"name","address","telephone"}=N_T

The last line is significant in that $T(N_T)$ = T(data types) = N_T. Note that this does not violate the axiom of regularity (x a set $\rightarrow$ x $\notin$ x). N_T is merely its own image under the function T; hence $N_T \in$ dom T and $N_T \in$ ran T, but $N_T \notin N_T$.

T is a function, but since D is finite, the range of T cannot be the entirety of $\mathcal{P}(D)$, i.e. T is not an onto function. Furthermore, T is generally not one-to-one. Apart from constraints of cardinality, the first condition will usually hold because not all members of $\mathcal{P}(D)$ form coherent data types in the eyes of the user. The lack of a one-to-one condition results whenever two data type names (or more) have the same image under T. For example, the set

S = {Babe Ruth, Bill Dickey, Tony Lazerri,
Lou Gehrig}

might be T(Murder's Row) or T(Old Yankee Sluggers). Note, however, that $T(name_1) = T(name_2)$ does not imply that $name_1 = name_2$, as in the above example.

The final element of structure within B is the mapping from N_R, a set of relation names and the set of relations existing among members of N_D. Before examining the formal definition of this mapping, we define the cartesian product (or cross product) over various members of the function T.

Let $I \subseteq N_T$, then the cartesian product of T over I is defined as

$$\underset{i\varepsilon I}{\times} T(i)=\{f \mid f:I\rightarrow D \;\&\; (\forall i)(i\varepsilon I\rightarrow f(i)\ \varepsilon\ T(i))\} \tag{3-4}$$

Thus the cartesian product is a set of functions, each of which is a set of ordered pairs; the following example illustrates:

$I=\{age,hair,eyes\} \subseteq N_T$

$T(age)=\{10,20\}$

$T(hair)=\{brown,blond\}$

$T(eyes)=\{blue,hazel\}$

$$\begin{aligned}\underset{i\varepsilon I}{\times} T(i)=\{&\{<age,10>,<hair,brown>,<eyes,blue>\},\\ &\{<age,10>,<hair,brown>,<eyes,hazel>\},\\ &\{<age,10>,<hair,blond>,<eyes,blue>\},\\ &\{<age,10>,<hair,blond>,<eyes,hazel>\},\\ &\{<age,20>,<hair,brown>,<eyes,blue>\},\\ &\{<age,20>,<hair,brown>,<eyes,hazel>\},\\ &\{<age,20>,<hair,blond>,<eyes,blue>\},\\ &\{<age,20>,<hair,blond>,<eyes,hazel>\}\}\end{aligned}$$

If $|S|$ denotes the number of elements in the set S, then

$$\left|\underset{i\varepsilon I}{\times} T(i)\right| = \prod_{i\varepsilon I} |T(i)|$$

A relation is defined as a subset of some cartesian product of T over a subset of N_T. The function R maps N_R

into the set of all possible relations among members of the data-base. Rigorously,

$$R=N_R \rightarrow \bigcup_{I \subseteq N_T} \wp\left(\underset{i \in I}{\times} T(i)\right) \tag{3-5}$$

Specifically, if $x \in N_R$, and R(x) is a relation over the cross product of T over some set I, $I \subseteq N_T$,

$$R(x) \subseteq \underset{i \in I}{\times} T(i) \tag{3-6}$$

Note that the set I may have several relations associated with the cross product in (3-6). For example, several different relations could be drawn from

$$\underset{i \in \{\text{name,address,telephone}\}}{\times} T(i)$$

yet all would be subsets of the same cartesian product. This implies that the exact contents of any relation cannot be established from knowledge only of the data type names involved.

The model B defined constitutes one of many possible models. However, the choice of any one model depends upon how well it represents reality, and more importantly, how well its behavior can be made to conform to the behavior of the situation it represents. From our data base, we wish to selectively retrieve certain pieces of information; hence, we require a set of operations which allow maximum control over model behavior. In addition to the cartesian product operation in (3-4) five other operations on members of R

will prove useful.

Let $x,y \in N_R$ and

$$R(x) \subseteq \underset{i \in I}{\times} T(i) \qquad I=\text{dom } R(x) \qquad I \subseteq N_T$$

$$R(y) \subseteq \underset{j \in J}{\times} T(j) \qquad J=\text{dom } R(y) \qquad J \subseteq N_T$$

Note that I=domUR(x) and J=domUR(y) are the data type names which participate in R(x) and R(y) respectively. The operations

$R(x) \cup R(y)$	union	
$R(x) \cap R(y)$	intersection	(3-7)
$R(x) - R(y)$	set theoretic difference	

are all defined if and only if I=J. Just as it makes no sense to add chickens to apples, the value of unioning a phone-book and a cook-book is moot within B. As with cartesian product, union and intersection are commutative and associative; set theoretic difference is neither.

Given a relation R (phone-book)

$$R(\text{phone-book}) \subseteq \underset{i \in \{\text{name, address, telephone}\}}{\times} T(i)$$

we may wish to retrieve only a portion of the stored information. To isolate a particular set S of name-address-phone elements, we need specify only those elements of R also in S. However, if we wish to restrict a relation via data types, we define a projection on R(x). Let X=domUR(x) and

let $Y \subseteq N_T$ be the projection set. Then $\Pi_Y R(x)$ is the projection of R(x) on those data types contained in $X \cap Y$. Formally,

$$\pi_Y R(x) = \{f: X \cap Y \rightarrow N_D \mid (\exists g)(g \in R(x) \ \& \ f \subseteq g)\} \tag{3-8}$$

For example $\pi_{\{name,telephone\}} R(phone\ book)$ would yield a relation containing all names with associated phone numbers; each name-phone number pair in the projection is a subset of at least one name-address-telephone number tuple in R(phone book). If $|domain \cup R(x)| = n$, then there are 2^n distinct projections. Thus for R(phone book), the projection sets {}, {name}, {name,address}, {name,telephone}, {name,address,telephone}, {telephone}, {address}, and {address,telephone} yield $2^3 = 8$ possible projections. Since f operates only on $X \cap Y$, members of Y not contained in X have no effect on the projection process. Thus while a subset operation on R(x) yields a particular set of functions on $dom \cup R(x)$, the projection operation takes from all functions in R(x) those ordered pairs whose left component is contained in the projection set Y.

Our sixth and final set-theoretic operation, that of relation composition, is the most powerful and the most difficult to define. Composition in GOLD STAR has two similar forms: composition in its simple form and composition under the auspices of a data type renaming transformation. Assume the existence of two relations--a phone-book relation

and a name-salary relation; from these we wish to assemble a third relation indicating salaries at various addresses. Our algorithm follows:

1. We note that R(phone book) $\subseteq$ T(name) $\times$ T(address) $\times$ T(telephone) and R(name-salary) $\subseteq$ T(name) $\times$ T(salary); each relation involves T(name) in the cartesian product of which each is a subset.
2. We take from each relation a member in which the names match and proceed to form a new relation in which members are subsets of T(name) $\times$ T(address) $\times$ T(telephone) $\times$ T(taxes).
3. The new relation will contain that subset of the 4-ary cross product above formed by following the rule in (2.).

Let X=types(R(x))=domUR(x), Y=types(R(y))=domUR(y), then the composition R(x)°R(y) is defined as

$$R(x)\circ R(y)=\{g:X\cup Y\rightarrow N_T \mid \exists f_\varepsilon R(x),\exists h_\varepsilon R(y)$$
$$((\forall t)(t_\varepsilon \text{dom } g \rightarrow$$
$$(t_\varepsilon \text{dom } f\rightarrow g(t)=f(t)) \&$$
$$(t_\varepsilon \text{dom } h\rightarrow g(t)=h(t)))\} \qquad (3\text{-}9)$$

Alternatively, each g=fUh. The simple composition operation composes on all those data type names which are found in both R(x) and R(y), and produces a new relation whose order is greater than or equal to the orders of the operand rela-

tions. Specifically if order (R(x))=|domUR(x)| and order (R(y))=|domUR(y)|, then new_order=order (R(x)) + order (R(y))-|X∩Y|.

A capability missing within simple composition is the ability to match data type names which are different, yet have intersecting images under T. For example, if R(registration) ⊆ T(student) × T(subject) and R(phone book)⊆ T(name) × T(address) × T(telephone), the R(registration)° R(phone book)={}=∅. Such a composition may be necessary if an instructor wishes to call all his students in a particular subject. To allow for this renaming, we define two functions λ and ρ which apply to R(x) and R(y) respectively:

$$\lambda: X \rightarrow N_T \quad , X = types(R(x)) = domUR(x) \tag{3-10}$$

$$\rho: Y \rightarrow N_T \quad , Y = types(R(y)) = domUR(y) \tag{3-11}$$

λ and ρ rename certain data types as specified by a user of the system. In the registration example, λ(student)="name" would yield a result ⊆ T(name) × T(address) × T(telephone) × T(subject); the dual result, (name)=student, would yield a result ⊆T(student) × T(address) × T(telephone) × T(subject). In both cases the information would be the same, differing only in the perception of involved individuals as "names" or "students". Formally, we define extended composition as

$$R(x)_{(\lambda,\rho)}^{\circ} R(y) = \{g: (ran\lambda \cup ran\rho) \rightarrow N_T \mid \exists f_{\varepsilon} R(x), \exists h_{\varepsilon} R(y)$$

$$((\forall t)(t \varepsilon dom\ g \rightarrow$$

$$((\forall u)(u \in domf \;\&\; \lambda(u)=t \rightarrow f(u)=g(t) \;\&$$

$$(\forall v)(v \in domh \;\&\; \rho(v)=t \rightarrow h(v)=g(t))))\} \qquad (3\text{-}12)$$

In a set theoretic data management system, the most powerful operation is composition, and as one would expect, is also the most complex to specify. Nonetheless, it is composition, simple and extended, which allows selective merging by data types of relations.

Reconstructability

The decision of what data types are to be involved in any relation is a non-trivial matter. In particular relations defined by (3-5) and (3-6) may be more "complex" than necessary. Assume for example, that a company records for each employee only his name, age, height, weight, employee number, salary, and years with the company. The relation R(employee) might then contain the member

{<weight,243>,<age,36>,<height,5'9">,<name,J. Smith>,
<salary,$22,000>,<years employed,9>,<man #,6532>}.

While all this data pertains to "J. Smith", certain other elements may have no particular bearing on each other; it is not clear, for example, whether a $22,000 dollar salary is significant vis-a-vis the height 5'9". However, the unique name J. Smith and the age "36" tell a certain fact about J. Smith. The projections $\pi_{(\text{name address})}R(\text{phone book})$

$\Pi_{(name,age)}$R(phone book), et cetera, have been composed with each other without giving us additional significance among the facts about J. Smith, except that such facts each belong with him; but this is known already if the functions (name,age),etc. are within our purview.

If we consider an office directory, however, the ability to obtain the original relation by composition of its projections may be absent. Let R(phone book)=

{{<name,Al>,<room,A-5>,<extn,317>},
{<name,Al>,<room,A-5>,<extn,318>},
{<name,Andy>,<room,A-5>,<extn,318>},
{<name,Andy>,<room,B-2>,<extn,442>}} (3-13)

No set of projections (save the trivial case $\Pi_{(name\ room\ extn)}$) can be composed to exactly reconstruct (3-13); spurious members would be coded.

Given $\pi_{\{name,room\}}$R(phone book)={{<name,Al>,<room,A-5>},
{<name,Andy>,<room,A-5>},
{<name,Andy> <room,B-2>}}

and $\pi_{\{name,extn\}}$R(phone book)={{<name,Al>,<extn,317 >},
{<name,Al>,<extn,318>},
{<name,Andy>,<extn,318>},
{<name,Andy>,<extn,442>}}

then composition would generate the spurious members {<name,Andy>,<room,A-5>,<extn,442>}, and {<name,Andy>,

<room,B-2>,<extn,318>}. In fact, no set of projections on (3-13), when composed, yields R(phone book). The ensuing theorem provides a condition sufficient for relation reconstruction. The phone book example in (3-13) is not reconstructable from any set of its projections because we can find no single projection which would uniquely "tie together" the information stored there. In the employee relation, there exists, among others, a binary function which maps every name to an age, i.e. there is a function from names to ages. This fact assures us that, given a projection containing a name, weight, etc., the composition of the name-age function and this projection will yield a member of the relation R(employee). The existence of such a function f gives rise to a definition of relation reconstructability:

Definition: Let $P=\{\pi_Y R(x) \mid Y \subseteq \text{dom} \cup R(x)\}$, i.e. the set of all projections on R(x). R(x) is reconstructable if and only if

$$\exists P' \subseteq P \text{ such that}$$

$$p' \overset{\circ}{\varepsilon} P', P' = R(x). \qquad (3\text{-}15)$$

With this definition in mind, we now state and prove a theorem sufficient (unfortunately not necessary!) for reconstructability:

Theorem.

If $\{S_1, S_2, S_3\}$ is a partition of dom R(x) and

$\pi_{(S_2 \cup S_3)}R(x)$ is a "function", that is, if

$$\forall \tau_2, \exists! \tau_3 (((\tau_2 \in \pi_{S_2}\pi_{(S_2 \cup S_3)}R(x)) \;\&\; (\tau_3 \in \pi_{S_3}\pi_{(S_2 \cup S_3)}R(x)) \;\&\; (\tau_2 \cup \tau_3) \in \pi_{(S_2 \cup S_3)}R(x)) \tag{3-16}$$

then R(x) is reconstructable, and

$$R(x) = \pi_{(S_1 \cup S_2)}R(x) \circ \pi_{(S_2 \cup S_3)}R(x) \tag{3-17}$$

Proof:

If Lemma I of Appendix A is applied, then in (3-16)

$$\pi_{S_2}\pi_{(S_2 \cup S_3)}R(x) = \pi_{S_2 \cap (S_2 \cup S_3)}R(x)$$

or

$$\pi_{S_2}\pi_{(S_2 \cup S_3)}R(x) = \pi_{S_2}R(x) \tag{3-18}$$

and

$$\pi_{S_3}\pi_{(S_2 \cup S_3)}R(x) = \pi_{S_3}R(x) \tag{3-19}$$

Thus the function (3-16) can be simplified to

$$\forall \tau_2 \; \exists! \tau_3 \; ((\tau_2 \in \pi_{S_2}R(x)) \;\&\; (\tau_3 \in \pi_{S_3}R(x)) \;\&\; (\tau_2 \cup \tau_3 \in \pi_{(S_2 \cup S_3)}R(x))) \tag{3-20}$$

We now show that 1) any member of the left side of (3-17) is also a member of the simple composition on the right, and 2) that conversely, any member of the composition is also a member of R(x). These two

steps constitute the proof of (3-17).

I. Let $g \in R(x)$. The definition of projection (3-8) assures the existence of some $f_1 \in \pi_{(S_1 \cup S_2)} R(x)$ and some $f_2 \in \pi_{(S_2 \cup S_3)} R(x)$ such that $f_1 \subseteq g$ and $f_2 \subseteq g$. That is

$$(\exists f_1, f_2 \subseteq g)(f_1 \in \pi_{(S_1 \cup S_2)} R(x) \;\&\; f_2 \in \pi_{(S_2 \cup S_3)} R(x))$$

Since dom $f_1 = S_1 \cup S_2$ and dom $f_2 = S_2 \cup S_3$,

dom $f_1 \cup$ dom f_2 = dom $g = S_1 \cup S_2 \cup S_3 =$ dom $\cup R(x)$.

f_1 and f_2 are functions; since the union of their domains equals the domain of g, $f_1 \cup f_2 = g$. The union of f_1 and f_2, however, is equivalent to the binary composition of f_1 and f_2, since

$$(\forall t \in \text{dom } g)(t \in \text{dom } f_1 \;\&\; t \in \text{dom } f_2 \rightarrow f_1(t) = f_2(t))$$

Thus the element $f_1 \circ f_2 = g$ for some $g \in R(x)$.

II. Let $f_1 \in \pi_{(S_1 \cup S_2)} R(x)$ and $f_2 \in \pi_{(S_2 \cup S_3)} R(x)$, such that

$$f_1 \circ f_2 \in (\pi_{(S_1 \cup S_2)} R(x) \circ \pi_{(S_2 \cup S_3)} R(x)).$$

f_1 is g restricted to $S_1 \cup S_2$ for some $g \in R(x)$.

Consider an f_2' which is g' restricted to $S_2 \cup S_3$

for some $g' \in R(x)$. Since $\pi_{(S_2 \cup S_3)} R(x)$ is a "function" in the sense of (3-16), f_2 restricted to S_2 = f_2' restricted to S_2, so $g = g'$. Hence, for all $f_1 \in \pi_{(S_1 \cup S_2)} R(x)$ and for all $f_2 \in \pi_{(S_2 \cup S_3)} R(x)$, there exists a $g \in R(x)$ such that $f_1\ f_2 = f_1 \circ f_2 = g$. Thus all elements of the composition of (3-17) are also members of R(x).

Q.E.D.

Knowledge that a relation is reconstructable can aid in retrieval operations on the data-base B. Given the relation

$$R(\text{phone book}) \subseteq \bigtimes_{i \in \{\text{name,address,telephone}\}} T(i)$$

and the necessity of telephoning someone named "John", then the operation

$$\pi_{\{\text{phone}\}}(\ R(\text{phone book}) \circ \{\{\langle \text{name,John} \rangle\}\})$$

will yield a unique phone number if John has only one phone. The above operation for any person would yield a unique phone number if there were a "function" in the sense of (3-16) from names to phone numbers. Note however, that if John has two phones then the above operation will yield a set of two elements.

Even if we have no function from names to telephone numbers, this does not imply that retrieval of a unique phone number is impossible. For example, if every pair of names and addresses is "mapped into" a unique phone number by the "function" of (3-16), then by specifying both John and the additional information that he is currently at 10 Main Street, we can retrieve his unique phone number. The operation

$$\pi_{\{\text{phone}\}}(\ R(\text{phone book})\ \circ$$

$$\{\{\langle\text{name,John}\rangle,\langle\text{address,10 Main Street}\rangle\}\})$$

will yield the unique phone number if the above condition holds.

2. Ordered Set Theory

In considering how we might implement a relation, we must come to terms with the most pervasive real-world consideration to which we are bound, namely order. Order is imposed in almost all environments; symbols and notation will be written in a linear order, and a representation of a set is free of order only by intention. Most extant computers employ a sequential numerical ordering scheme, the most common of orders. Hence, we must conform to this constraint of order and include this notion somehow within our framework of set-theoretic concepts.

An ordering Ω is a bijection (one-to-one, onto mapping) from an index set I to a set S, each set containing k elements. The index set I contains the successor of all but one of its elements.

$$\Omega: I \; S \text{ where } \exists_{j \in I} | j^+ \notin I \; \& \; \forall_{i \neq j, \, i \in I} \; i^+ \in I \qquad (3\text{-}21)$$

Paraphrased, Ω assures us that we can step in some well defined sequence over the index set (via successors). Elements of S are thus associated with an ordering procedure.

With <a,b> defined as {{a},{a,b}}, we define a special case of ordering, the tuple. We already have an intuitive feel for a tuple, e.g. (a,b,c). We recognize that the physical position of an element is as much a piece of necessary information as is the element itself. The tuple (b,b) can exist, for example, while the set {b,b} cannot. But a tuple can be viewed as follows:

A tuple τ is an ordering Ω_T where the index set I_T has the special properties that it is a subset of the natural numbers and contains the element 1 if non-empty. Formally

$$\tau: \Omega_T: I_T \rightarrow S \text{ where } I_T \subseteq N, \exists_{i \in I_T \rightarrow 1 \in I_T}, \exists_{k \in I_T} | \; k^+ \notin I_T \; \& \; \forall_{i=k, \, i \in I_T \rightarrow i^+ \in I_T} \qquad (3\text{-}22)$$

That is, a tuple is the special ordering from {1,2,3,...,n} to the n elements of the range set. In set notation, a tuple is represented as

$$(t_1,t_2, \ldots ,t_n) = \{<1,t_1>,<2,t_2>, \ldots ,<n,t_n>\}$$

A tuple is readily represented on a real computer, since the order of a tuple coincides with the most practical implementations of order. However, an arbitrary ordering Ω may not be easily representable; therefore, at some point, if no other assumptions are made, we must use tuples (or whatever real-world form of ordering to which we are bound).

As an alternative to a given environmental ordering we define an entity called an explicit element.

> An Explicit Element e is an element which is 1) meaningful in a global sense, and 2) recognizable independently of the context in which it is found.

For instance, if the word "fremitus"[3-2] appeared only once on this page, we could locate it unambiguously, independent of the way in which the words on the page are searched. We can also determine its meaning free of the context of this page. Note that any element can become explicit by merely restricting the context of consideration

sufficiently; if we search only one word it must be explicit since no duplication exists as long as the word is meaningful. The requirement of global meaning is significant in computers, since many words are not meaningful to a given search request (e.g. the octal word 777777 has no meaning when viewed as four ASCII characters.)

Often we wish to make the elements of N_T and N_R (see section 1. of this chapter) explicit elements. It is desirable that these names be recognized in a context-free sense whenever they appear. Consider the set E of explicit elements where $e \varepsilon E \rightarrow e \varepsilon N_T$. With E we define another entity which will be reencountered later in the discussion.

> An Explicit Naming Tuple is a tuple whose range consists entirely of elements e, where $e \varepsilon N_T$ and e is an explicit element.
>
> ENT: $\Omega_T : I_T \rightarrow \{e_1, e_2, \ldots, e_n\}$ $e_i \varepsilon N_T, e \varepsilon E$ (3-23)

In effect, an ENT defines an arbitrary order on the elements of E which will prove useful. Since ENT is a function, its range is totally ordered.

The ENT function is intimately tied to the definition of a relation. A relation R(x) assumes the form

$$R(x) = \{\{<e_1, d_{11}>, <e_2, d_{21}>, \ldots, <e_n, d_{n1}>\},$$

$$\{\langle e_1,d_{12}\rangle,\langle e_2,d_{22}\rangle, \ldots ,\langle e_n,d_{n2}\rangle\},$$
$$\{\langle e_1,d_{13}\rangle \langle e_2,d_{23}\rangle, \ldots ,\langle e_n,d_{n3}\rangle\}$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$\{\langle e_1,d_{1m}\rangle,\langle e_2,d_{2m}\rangle, \ldots ,\langle e_n,d_{nm}\rangle\}\} \qquad (3\text{-}24)$$

Since R(x) is a set, the subscripts identify, but do not order the particular elements involved. Note that

$$\text{dom} \cup R(x) = \{e_1,e_2, \ldots ,e_n\} \qquad (3\text{-}25)$$

Consider the class of functions

$$\text{ENT: } \{i \mid 1 \leq i \leq n\} \rightarrow (1 \text{ to } 1, \text{ onto}) \text{ dom} \cup R(x) \qquad (3\text{-}26)$$

There are n! possible ENT functions for any R(x); each such ENT will be denoted by a subscript. We take in particular some ENT_k such that

$$ENT_k(R(x)) = \{\langle 1,e_1\rangle,\langle 2,e_2\rangle, \ldots ,\langle n,e_n\rangle\}$$
$$= (e_1,e_2, \ldots ,e_n) \qquad (3\text{-}27)$$

The subscript k serves only as an identifier for the particular ENT chosen. An equally valid choice might be

$$ENT_{k'}(R(x)) = (e_n,e_{n-1},e_{n-2}, \ldots ,e_2,e_1) \qquad (3\text{-}28)$$

$ENT_k(R(x))$ and $ENT_{k'}(R(x))$ are distinct bijections on the same relation. Each ENT function is an arbitrary ordering of the data type names in dom $\cup R(x)$. The particular order is specified by which of the n! possible orderings is chosen.

As defined in (3-27), an ENT is a function with respect to a relation. The particular k chosen depends on the organization of R(x) desired. This indicates how the ENT empirically defined in (3-24) is chosen. The explicit nature of the elements of dom$\cup$R(x) is, of course, maintained.

At this point, we introduce some additional functions which transform R(x) into another entity. The transformations will also exhibit the following characteristics of the relation R(x).

1. Given a data element d N_D, we can determine a type with which it is associated. This is possible since each member of a relation R(x) is itself a bijection:

   $$t: e \rightarrow d \mid t \varepsilon R(x) \;\&\; (\forall i,j \leq n)\ \text{dom}\ t_i = \text{dom}\ t_j \qquad (3\text{-}29)$$

   t^{-1}, in turn, maps d onto e. Note that since t is a bijection, we are always a unique e given a specific d.

2. Given an e and an R(x) we can determine all instances of d (i.e. the exact subset of N_D) which correspond to that e. These are merely the mappings t over all of R.

3. Every element of R(x), by virtue of the fact that

R(x) is itself a set, indicates a relatedness among its members. That is,

row: x "is related to" y where $x,y \varepsilon \text{ran } t$, $t \varepsilon R(x)$ (3-30)

4. Every member of R(x) includes an ordered pair for every e. The d's in these pairs are also related by virtue of their association with the same type e.

column: x "is related to" y where $x=t_i(k)$, $y=t_j(k)$ for all $i,j \leq m$ and for all $k \leq n$ (3-31)

The nature of relatedness expressed in characteristics three and four is at this point intuitive, but will become clearer as transformations are performed upon members of R. The names "row" and "column" are used to indicate that when notation such as in (3-24) is used, relatedness exists by virtue of the physical structure of rows and columns.

We define a function called Tuple with Explicit Naming Tuple (TENT):

$$\text{TENT: } t \rightarrow t' \mid t \varepsilon R(x),\ t' = t \circ \text{ENT} \qquad (3\text{-}32)$$

More specifically, consider a given $ENT_k R(x)$. Then

$$TENT_k(t) = t \circ ENT_k(R(x)) \mid t \varepsilon R(x) \qquad (3\text{-}33)$$

Note that both the domain and range of $TENT_k$ are functions. The domain is a function from e to d; the range is a tuple.

Using the notation for R(x) in (3-24), we can formally express a typical TENT:

$$ENT_K=(e_1,e_2, \ldots .e_n) \quad [k \text{ arbitrary}] \tag{3-34}$$

$$\begin{aligned} TENT_k(t_1)&=\{<e_1,d_{11}>, \ldots ,<e_n,d_{n1}>\} \circ \\ &\quad \{<1,e_1>, \ldots ,<n,e_n>\} \\ &=\{<1,d_{11}>, \ldots ,<n,d_n>\} \\ &=(d_{11},d_{21}, \ldots ,d_{n1}) \end{aligned} \tag{3-35}$$

We note that the elements of the TENT are ordered in a fashion analagous to that of the particular ENT_k. If k' from (3-28) were used instead, the TENT would appear as:

$$TENT_{k'}(t_1)=(d_{n1},d_{n-1,1}, \ldots ,d_{21},d_{11}) \tag{3-36}$$

It should be clear that each of the TENT's (with respect to a given relation R(x)) will each be ordered with respect to the ENT_k chosen to order the given R(x), since

$$\forall i,j \; 1\leq i,j\leq m \quad \text{dom } t_i=\text{dom } t_j=\text{ ran ENT} \tag{3-37}$$

In general then, for the R(x) of (3-24) the i^{th} TENT:

$$\begin{aligned} TENT_k(t_i)&=(d_{1i},d_{2i},d_{3i}, \ldots ,d_{ni}) \\ TENT_{k'}(t_i)&=(d_{ni},d_{n-1,i},d_{n-2,i}, \ldots ,d_{1i}) \end{aligned} \tag{3-38}$$

and so on for any choice of ENT.

Note that we preserve the same content between a $TENT_k(t_i)$ and its ENT_k as with the element t_i of R(x) itself. This follows since all derivations are bijectional,

and it is possible to "go back". Thus we seek a structure which will include all $TENT_k$'s and the ENT_k. This structure would then contain all the information found in the relation R(x) itself.

Therefore we define a sort on the various TENT's

$$SORT(R(x)):\{j \mid 1 \leq j \leq m\} \rightarrow (1\text{-to-}1,\ \text{onto}) \quad \{TENT(t) \mid t \in R(x)\} \qquad (3\text{-}39)$$

There are m! possible ways of ordering the m TENT's derived from R(x). SORT is esentially a second arbitrary order placed upon the relation. However, there are a few standard SORT mappings which are used more frequently than others. The one used in GOLD STAR is called a <u>lexicographic</u> sort. We define lex ordering of TENT's as follows:

$$TENT_k(t_x) \prec (\text{lex less than})\ TENT_k(t_y) \text{ iff}$$
$$(\exists i)(1 \leq i \leq n\ \&\ (\forall j)(j < i \rightarrow TENT_k(t_x)(j) = TENT_k(t_y)(j))\ \&$$
$$TENT_k(t_x)(i) <^* TENT_k(t_y)(i) \qquad (3\text{-}40)$$

where

$$TENT_k(t_\zeta)(p) = \text{the } p^{th} \text{ element of the } \zeta^{th} \text{ TENT}$$

Note that the $<^*$ operation is determined by the ENT_k chosen in the sense that the i^{th} element of the TENT is determined by the orderings ENT_k. For example,

Let

$$TENT(1)=(1,2,1,3)$$
$$TENT(2)=(1,1,1,1)$$
$$TENT(3)=(2,1,3,3)$$
$$TENT(4)=(2,1,3,2) \qquad (3\text{-}41)$$

be determined a relation

$$SORT_{lex}(\text{the TENT's}) = (TENT(2), TENT(1), TENT(4), TENT(3))$$

For example, then,

$$TENT(4) \prec TENT(3) \text{ since for } i=4,\ j=1,2,3$$
$$TENT(4)(1) = 2 = TENT(3)(1)$$
$$TENT(4)(2) = 1 = TENT(3)(2)$$
$$TENT(4)(3) = 3 = TENT(3)(3)$$
$$TENT(4)(4) = 2 < 3 = TENT(3)(4)$$

Lex ordering, then, is the intuitive order where the first component of the TENT's is most significant, the second is next most significant, etc. The importance of ENT_k is in the determination of <u>which</u> is the first component, which is the second, and so on.

We now define what appears as a trivial construction, the QUART:

$$QUART = (ENT_k(R(x)), SORT(R(x))) \qquad (3\text{-}42)$$

(QUAsi RelaTion)

For a typical relation R(x) as defined in (3-24),

$$\begin{aligned} &\text{QUART}(R(x)) = \\ &((e_1,e_2, \ldots ,e_n),(d_{11},d_{21}, \ldots ,d_{n1}), \\ &\qquad\qquad (d_{12},d_{22}, \ldots ,d_{n2}), \\ &\qquad\qquad \vdots \quad \vdots \qquad\quad \vdots \\ &\qquad\qquad (d_{1m},d_{2m}, \ldots ,d_{nm})) \end{aligned} \tag{3-43}$$

The construct QUART has two noteworthy properties, over and above those possessed by R(x)) itself:

1. Two arbitrary orders are imposed upon the relation, namely $\text{ENT}_k(R(x))$ and SORT(R(x)).
2. The elements of ENT appear only once, giving a "factored" effect. This is highly desirable since by the definition of explicit elements, we can now consider a QUART as a context of consideration. (See 3-23). The relation in its proper form has redundant occurances of the explicit elements, which limits our context. When transformed to a QUART, we can unambiguously identify the e's, and hence ease the search for information.

Note that QUART(R(x)) retains all the information found in R(x) and discussed previously. By following the various bijections created in the derivations we can freely translate one form of information to another, e.g.

1. Given a d, find an e, $e_i = ENT_k(TENT_k^{-1}(j)(d_{ij})$
2. Given an e, find the d's associated with it

   $d' = \{TENT_k(j)(ENT_k^{-1})$ over all j's$\}$
3. Each TENT relates all d's with it.
4. The d's defined above relate all d's of a given data type.

The notation used in (3-43) lends itself to an even clearer form:

$$\begin{array}{cccc} \underline{e_1} & \underline{e_2} & \cdots & \underline{e_n} \\ d_{11} & d_{21} & \cdots & d_{n1} \\ d_{12} & d_{22} & \cdots & d_{n2} \\ \vdots & \vdots & & \vdots \\ d_{1m} & d_{2m} & & d_{nm} \end{array} \qquad (3\text{-}44)$$

This form is already familiar from (2-1) and (2-2-d). As expected this form represents a relation.

Henceforth, where a QUART is mentioned, it will be assumed that it is in effect a relation. We refer to its structure as follows:

CENT, or PENT: Two computer representations for the types, or e's in our formulation.
(CENT is Character ENT).
(PENT is Pointer ENT).

Order: The number of elements in the ENT and each TENT (= $|\text{dom} \bigcup R(x)| = n$).

Length: the number of TENT's in the QUART, which was m in our formulation.

When we mention the term "relation" henceforth, we mean a relation stored in some form other than a QUART. This section has discussed how any representation of a relation (which represents the abstract notion of a relation in (3-24) can be transformed into a QUART. Other relational organizations are considered in the next chapter.

One final concept remains: that of a successor in the context of a QUART. We have adopted the view that the successor of a TENT is the next TENT according to the SORT function adopted. This follows either the specified TENT, if it is part of the QUART, or follows the place where the TENT would have been inserted if not already included. More succinctly, the successor operation is a least upper bound operation. For example, given

a	b	c
1	1	1
1	3	1
1	3	3
2	1	2

successor[(1,3,1)] = (1,3,3)

successor[(1,2,1)] = (1,3,1)

Further examples of QUART operations are found in

Appendix I.

With the theoretical concepts of GOLD STAR now enunciated, it will prove interesting to examine the structure and implementation methods for a set-theoretic approach to data-management.

CHAPTER IV

AN IMPLEMENTATION OF GOLD STAR

This chapter discusses the rationale for one GOLD STAR implementation. The system is not of the direct transplant variety, since we have chosen to utilize some features peculiar to the present target computer -- the GE/645/MULTICS. Furthermore, the algorithms mechanized for the MULTICS system may be less suited to environments other than ours. Despite certain shortcomings, we feel it advantageous to pursue the cogs and machinery of GOLD STAR/MULTICS.

1. DSM Modules

Representation of Data Elements

Every computer system faces the problem of dual representation of data elements: one representation allowing fast internal operation of the programs, and another suitable for human consumption via printed or graphic input/output operations. Classically, i.e. in "algorithmic" languages, the formats for user data have been dictated by "representation types" such as integer, floating point, logical, and fixed point data found in FORTRAN systems. In GOLD STAR the term "data type" refers to classification of data according to some attribute assigned by the user (e.g. "cities","last name"), and not according to some internal code peculiar to the computer environment. To the user, this distinction between "representation type" and "data type" is significant:

in the former case, environment idiosyncrasies force extra meaning on the data due to representation, while in the latter case all semantics depend upon the user's own classification of the data.

While the user views his data elements only in terms of data type, there still remains the issue of efficient internal representation. Although relatively transparent to the user, the techniques utilized to increase the speed of operation merit further scrutiny. Internally, all user data is manipulated in the form of word-length bit string tokens, which we refer to as reference numbers. The task of binding strings to reference numbers is performed by a class of GOLD STAR sub-programs called data strategy modules, or DSM's.

The Binding of Reference Numbers to Strings

At the time a user creates a new data type within GOLD STAR, he usually perceives or assumes a total ordering on the elements of a data type. The method of assigning reference numbers to strings begins with the requirement that

> *The total ordering on members of any data type is completely preserved by (i.e. isomorphic to) the total ordering on the assigned reference numbers.*

We further require that valid comparisons between data items exist only if the comparands are numbers of the same data type.

Binding of Reference Number Via Alphabetic Order

The module dsm_astring binds strings to reference numbers in such a way that ascending reference numbers preserve alphabetic order among the data elements. The notion of alphabetic order is deeply ingrained in most individuals, and is significant in that it is a common ordering scheme applicable to arbitrarily large lists of strings of any length(s). Universal usage of alphabetic order requires a facility, i.e. dsm_astring, for its accomodation; dsm_astring serves this purpose.

For each data type it manages, dsm_astring maintains a binary tree in which data elements are stored and from which they are retrieved. Because collating sequences in ASCII and otner codes preserve alphabetic order, dsm_astring is able to compute order-preserving reference numbers directly from the input character strings. The exact method by which this operation works will become clearer if we follow the procedure invoked to perform a binding operation on the data type "names". (By binding we mean inserting a new item into a DD area and associating an appropriate reference number.)

The user creates the data type "names" via a call to mgr$new_data_type and initializes the header shown in Figure 4-1. (Each data type is stored in a separate segment in the MULTICS system, and its size is controlled via the

"allocate" and "free" statements in the PL/1 language.) The header contains information required by GOLD STAR for system calls, as well as pointers to the tree root, the head of a successor chain, and counters used to indicate the number of free and empty cells. The user establishes to GOLD STAR that the DD area "names" will be managed by dsm_astring; he does so by specifying the DSM name in the call to new_data_type. Subsequent references to "names" will automatically invoke this DSM.

Figure 4-2 illustrates the logical changes in "names" resulting from the insertion of "Ezra". Two actions occur: the root pointer is set to the address of "Ezra", and the canonical root reference number is assigned. The root reference number is defined as

$$ref_{root} = 2^{n-2}$$

where n = the number of bits in a full word. The choice of this quantity as the root reference number implies there are as many positive reference numbers greater then ref_{root} as there are less than ref_{root}, and facilitates the formation and maintenance of a tree symmetrical about the root. In subsequent action, the successor chain head is set to "Ezra" and the number of cells active in "name" is incremented by "1". As "Ezra" has neither descendants nor a successor at this point, these pointers in the data item are null.

Figures 4-3 and 4-4 illustrate the method by which dsm_astring computes reference numbers. In Figure 4-3 the name "Claude" is inserted in the tree as Ezra's left descendant, since "Claude" is lexicographically less than "Ezra". The reference number of "Ezra" (= ref_{root}) is the only reference number at level 1 in the tree. Reference numbers for descendants of an item present in the tree are computed as follows.

A candidate data item for insertion is compared with the root note item. If the candidate is less than the root (according to alphabetic ordering), then one of three situations can occur. If the root item and candidate are equal, then the candidate is already present; search stops. If the root item's left descendant is non-null, then it becomes in effect the root, and search begins at that point. If the root item's left descendant is null, then a number of words required to store an item are allocated, pointers are chained and the new reference number is computed via the formula

$$r_{left\_descendant} = r_{parent} - 2^{(d-i)}$$

where d is the maximum depth of the tree, and i is the depth of the tree at which the parent node appears.

If the candidate item is alphabetically greater than the root item, then a procedure analagous to the above procedure for left descendants is executed, on the right, with

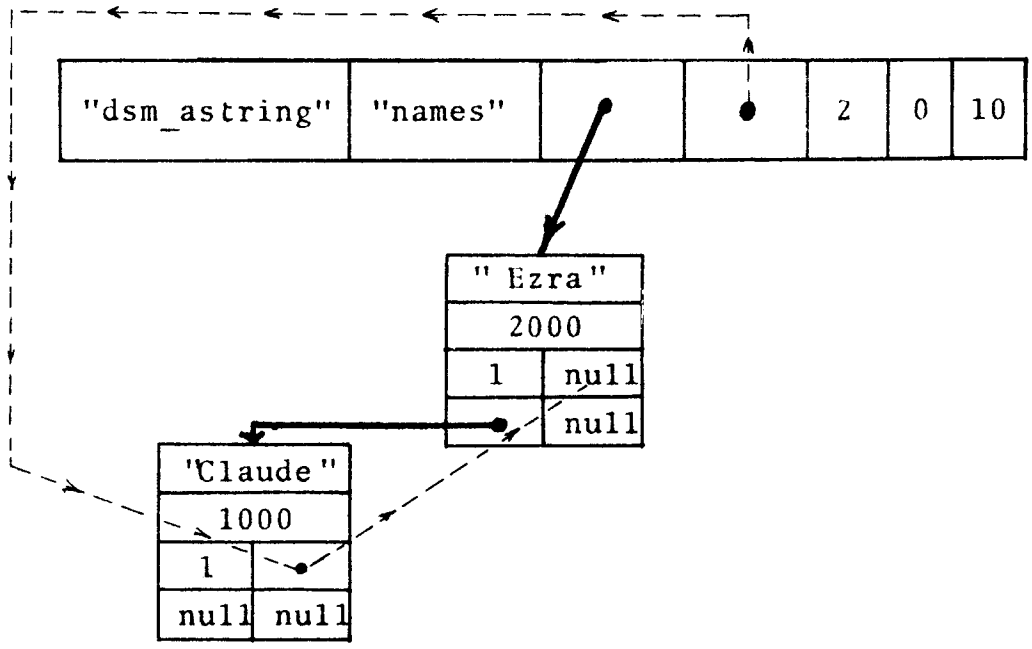


Figure 4-3: Insertion of "Claude" in Data Area

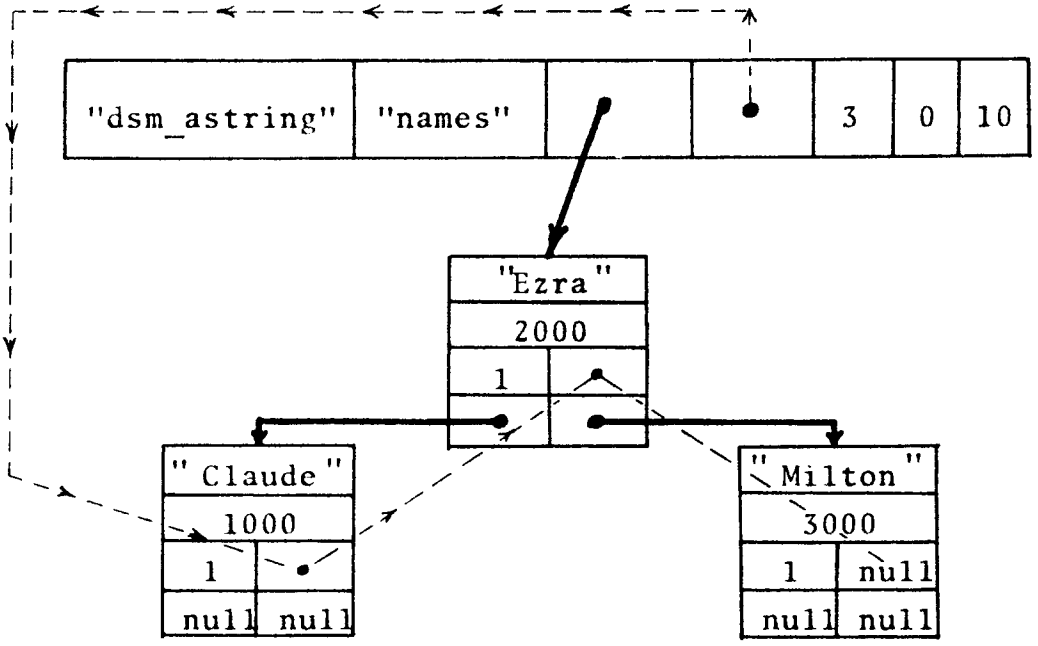


Figure 4-4: Insertion of "Milton" in Data Area

$$r_{right\_descendant} = r_{parent} + 2^{(d-i)}$$

Note that the reference numbers for left and right descendants will be less than and greater than, respectively, the reference number of the parent node as long as $i \leq d$. When i exceeds d, this implies that no more items may be inserted in the tree below the current "working node". The exact reasons for this limitation result from the fact that the level of the tree i occupies one bit position in the reference number, with a left turn at level i being indicated by a 0, a right turn by 1 in bit position i in the reference number. Thus a reference number of d bits allows at most d levels to a binary tree. A more complete discussion of dsm_astring and tree-overflow is found in Appendix A.

The module dsm_astring is ideally suited to the task of binding reference numbers to alphabetically ordered strings, since insertion and deletion of items requires no reshuffling of data elements. The ability to delete and insert items without restructuring is an important consideration, but perhaps the greatest value of the tree search is the fact that for a data type of n members, the average search time is $K\ell og_2 n$ where the constant K is independent of n.

Binding of Reference Numbers to Integers

Like the ordering on alphabetic strings, an ingrained ordering is the consecutive nature of the integers. Classi-

cal data management systems have recognized this important ordering, but only by providing a specific internal computer representation. As mentioned at the beginning of this section, GOLD STAR treats all data types in relation only to a perceived ordering. Thus a module dsm_integer maps data elements representing integer ordered strings into fixed length bit strings. (The dsm_integer module uses the same internal format for integers that a classical data management system would. That is, strings made up of the characters "0" to "9" are transformed into internal machine binary representation. Thus, GOLD STAR converts character strings representing integer data, to binary numbers representing integers to the target computer. However, "integer" is not a data type; data types such as "population", "number_of_children" etc. are examples of data types utilizing the integer ordering on strings.)

The Binding of Reference Numbers to Strings Where a Well-Ordering of String is Inapparent from the Alphabetic Order

Two other data strategy modules complete the data representation repertoire of GOLD STAR. Very few data types of large membership ever utilize an ordering scheme other than the two previously mentioned, simply because humans find memorization of many ordering schemes inconvenient. However, certain data types of small membership size (e.g. the months, or academic titles) follow ordering relations not directly discernible from the data elements.

In cases where the ordering is static (i.e. no insertions in the data type, as in the case of "months of the year") the module dsm_table assigns indices in the table to data elements. The data type "months of the year" would map January in 1, February into 2, etc. For small data types which are of dynamic rather than static nature, (e.g. the collection to titles in a company) a module dsm_chain is provided. dsm_chain allows insertion and deletion of items which are maintained in order on a single threaded list. An inserted item assumes the reference number

$$r = \frac{1}{2} (r_{predecessor} + r_{successor}).$$

2. RSM Modules

Every relation in GOLD STAR is stored in its own segment, and each such segment is organized and managed by a program known as an RSM, or relational strategy module. An RSM operates on one particular physical organization of relations; in most instances the number of relations will far exceed the number of separate relation organizations.

The choice of which RSM is the most likely candidate for creation and maintenance of any relation is probably the user's greatest single problem. He must decide which structure is most suited to operations involving the relation. If the relation is large, will it require simple organization due to storage limitations? Will the relation be modified frequently? Is there more than one frequently

used lexicographic sort of the relation?

Among these questions, there is a hint of the magnitude of problems involved in the choice of an RSM for any relation. Like the DSM choice, several factors are involved, and since most CPU time will be incurred during RSM operations, the judicious choice of an RSM is critical. The more significant factors are:

A. The number of different lexicographic sorts with respect to which the relation will be accessed.

B. The frequency of use of the relation.

C. The probability distribution expressing relative access to each item.

D. The number of tuples in the relation.

E. The envisioned dynamic nature of a relation. If insertion and deletion of relation members is frequent, then some RSM which chains items together by some ordering other than storage location is called for.

F. The basic structure of the relation itself. A many-to-one or one-to-many relation may have to be managed entirely differently from a nearly one-to-one relation. For example, a relation in which names have several telephone numbers is many-to-one.

Thus far, these guidelines have been enunciated, but very little is known (other than empirical testing) about their measurement. Our efforts this past year have provided an RSM managing quarts, and other possible RSM structures. We decided that all RSM's, in addition to normal operation entries, should contain entry points for converting between a quart and the particular relation structure managed, and vice versa. This allows a user to consider several alternative RSM's and test the operation of each; the user enters his relation once, and can then proceed among the desired RSM's via subsequent calls to convert_to_quart and convert_to_relation. Morever, this double structure format involves only a small amount of code compared to most operations.

The following constitute the RSM repertoire in the initial version of GOLD STAR.

A. RSM_Q -- assumes operands are quarts, but treats them as non-transient members of the function R.

B. RSM_WQ -- assumes operands are quarts, but does not provide for keeping these quarts beyond the life of a process. (See next section).

C. RSM_TREE (under consideration)

While both RSM_Q and RSM_WQ manage extremely simple relation organizations (which are lexicographically ordered matrices with some trimmings) they offer the least power of

all RSM's. Sorting, dynamic insertion, and dynamic deletion all require creation of new quarts which hold the modified relation. In cases where large relations are involved, these operations, or other RSM operations requiring these services as subfunctions (composition, union, intersection, etc.) may become extremely costly and time-consuming. Moreover, in on-line operations, searching for or modifying even a single tuple in large relations may disallow quick real-time response. For certain relations, it may prove expeditious to use a tree structure similar to those of TDMS or the MacAIMS project at M.I.T. While these structures require more space to implement, addressing an item depends not upon restructuring, but rather upon the use, manipulation, and modification, of certain pointers at each node.

The RSM_WQ

This is the module which performs the basic system operations on quarts only. The reader is referred to Appendix I for some examples of such operations. A few interesting issues have arisen in the implementation of the Working Quart RSM.

A. Union, Intersection, Difference

These operations all use fundamentally the same code (see Figure 4-5 for flow chart). A key issue, however, was whether to sort the second input with respect to the

first before operating, or not. To solve this dilemma,

Let ℓ_1 = length of first quart

ℓ_2 = length of second quart

0 = order of each quart

S = a "search", or the comparing of one quart element with some given entity.

We know that a simple token sort of the kind used in RSM_WQ takes a number of steps

$$\#S(\text{sort}) = 0^2\ell \log\ell \text{ (on the average)}$$

Once the quarts are both sorted alike, we can recognize a best case in operating.

When the first TENT of one quart is lexicographically greater than the last TENT of the other, then it is possible to determine this by only examining the first element of each of those TENTS; we can then determine the union, intersection or difference. For example, given

a	b	c
1	2	3
4*	5	6

a	b	c
10*	11	12
13	14	15

since 10 is greater than four, we know the intersection is empty, the difference is the first quart, and the union is the combination of all TENT's of the two.

We can also recognize a wors case (see flow chart) where we must examine every element at least once during

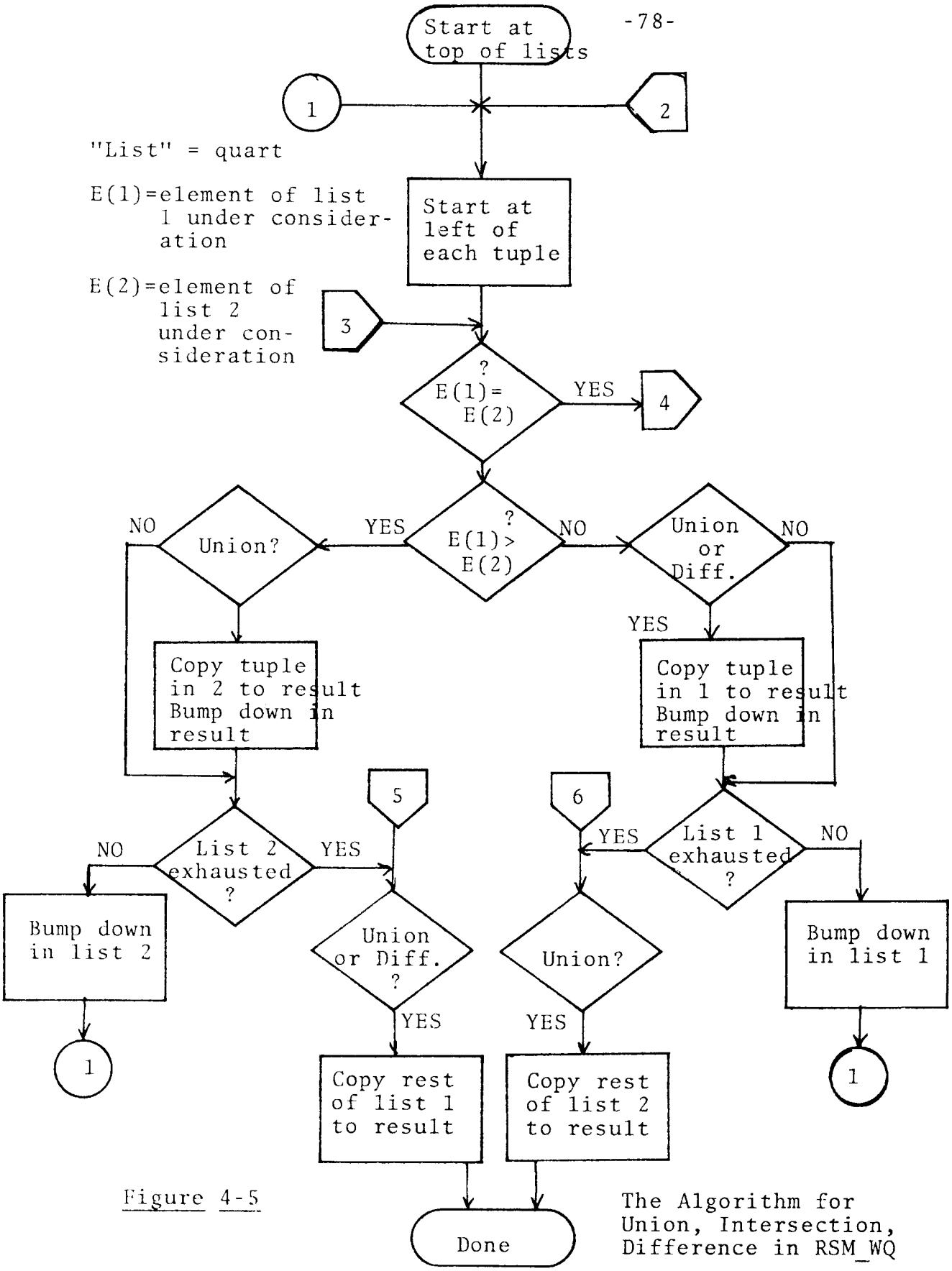


Figure 4-5

The Algorithm for Union, Intersection, Difference in RSM_WQ

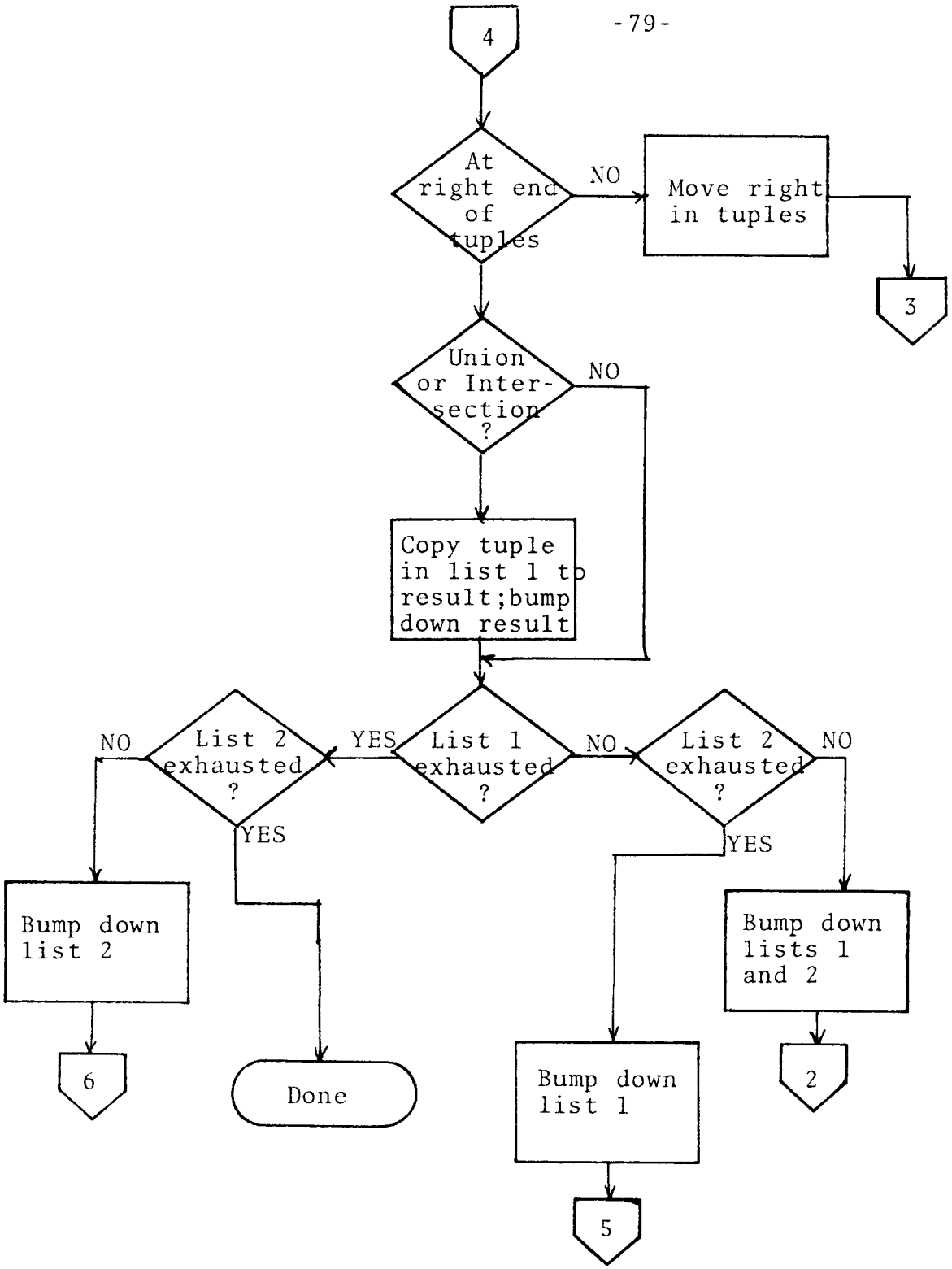


Figure 4-5(Continued) -- The Algorithm for Union, Intersection, Difference in RSM_WQ

the steady progression through each quart. This yields

$$2 \leq \#S(\text{operate}) \leq (\ell_1 + \ell_2)0$$

Hence, to sort and operate, we get, as an average,

$$\#S(\text{sort \& operate}) = (\ell_1 + \ell_2)\frac{0}{2} + 0^2\ell_2 \log \ell_2$$

In assessing just how many comparisons are needed to operate in an unsorted case, we again examine a best case:

> For each TENT in quart 1, we need only search the entire length of quart 2 once ($\ell_2/2$ average searches) during which we find an equal TENT; that means that the first match found for each of the TENT elements is the only one. An example of this would be:

quart 1:	a	b	c	2:	b	c	a
	1	2	3		2	3	1
	4	5	6		10	11	12
	7	8	9		5	6	4

> Here, if we start searching the a column in the second quart, we need go no further since the b and c columns will also (and always) match (presuming a match was found at all in a). This best case requires $\ell_1(\ell_2 0)/2$ searches.

We can also ascertain a worst case:

> For each TENT in quart 1: every column in quart 2 must be searched ($\ell_2 0/2$ average searches) for each of the columns in quart 1. That is to say, when a match

is found in column a, one is not found in b and a must be searched for again, with the worst case being once for the entire length of a (if all of a is the same). This case takes $\ell_1\ell_2 0^2/2$ searches on the average.

Let us now compare the two cases:

Sort then operate	No sort
$2+0^2\ell_2\log_2 \leq \#S \leq (\ell_1+\ell_2)0+0^2\ell_2\log\ell_2(\ell_1\ell_2 0)/2 \leq \#S \leq (\ell_1\ell_2 0^2)/2$	
On the average	
$(\ell_1+\ell_2)0/2 + \ell_2 0^2\log\ell_2$	$\ell_1\ell_2 0^2/4$
letting $\ell_1=\ell_2=\ell$ approximately	
$\ell 0 + \ell 0^2 \log\ell$	$\ell^2 0^2/4$
dividing each by $\ell 0$	
$1 + 0\log\ell$	$\ell 0/4$
for $\ell=100$, $0=3$	
$1 + 6 = 7$	75
for $\ell=10$, $0=3$	
$1 + 3 = 4$	7.5

So, for most configurations of quarts, the sort operation is well worth the tune. That is why that course of operation was chosen.

B. Composition

A first consideration in composition is how to specify it. We face a number of decisions:

1. Compose or not on columns with equal CENT's
2. How to indicate composition on columns with unequal CENT's
3. Whether or not to keep the resulting column

The scheme adopted of binding all CENT's involved to a user supplied vector of names has the following effects on the decisions:

1. We compose by equality of bindings, not CENT's. This allows a choice on columns with equal CENT's, since they can easily be bound to themselves.
2. By binding unequal CENT's to equal names, we can choose this alternative.
3. This is not optimal; all columns are kept. Projection will elimanate unwanted columns.

The equivalence vector also presents the easiest form to deal with on a practical level; a complete equivalence relation indicating classes of equivalence by CENT would be very difficult to specify syntactically.

The equivalence vector for $R(x)_{(\lambda,\rho)}^{\circ}R(y)$ (see Chapter III) is the concatenation of $ENT_K(\lambda)$ and $ENT_{K'}(\rho)$ where K and K' are the sorts specified by the user. To

output_expression)

where the input expression is a regular expression which is "matched" to the input, and the output expression dictates how to transform the bindings to an output string. The regular expression syntax considered is as follows:

Delimiters are: blank, and parentheses

Operators are: or (|), closure (*), not (^), and concatenation (e.g. a|b matches a or b, a* matches any number of a's, ^a matches anything but a, ab matches ab).

Special characters are: · (matches any character), $ (matches any number), ? (matches any upper case), ! (matches any lower case), < (matches the beginning of a line), > (matches the end of a line).

Bindings are made by /numeric, variable/ and are created only if the numeric expression is satisfied. The format of the numeric is n+ (n or more characters in the string to be bound), n- (n or less), and combinations of these using concatention or or.

Some examples are:

RE	Input	Bindings
<a*|b*/5-,x/>	aaamp	x → mp
same	aamnopqr	none (will not match)
a|bc/x/$#	bcmno231	x → mno

The user is referred to explanations of QED for further elucidation, and to Appendix C for a formal syntax definition.

The main drawback to implementing such a parser is the lack of efficiency that code doing these tasks will have. It is necessary to, in effect, compile machine code from these regular expressions and execute that code. No attempt has yet been made to implement this design, although it is hopeful that the task is not too complex to undertake.

Protection and The Projector (See Chapter II, Figure 2-3 for the part the projector plays in the system structure)

A situation that is sure to arise is that of having privileged information which is not isolatable to a single data area or relation. For instance, the relation name-salary-contract-percentage might exist; salaries are confidential while contract and percentage are not. In effect, we are in the position of wanting to allow access to a projection of a relation. Given the MULTICS protection scheme (see Appendix D for a full discussion of protection issues), a need for a "projector" arises. This projector would operate in a ring higher than that of the user. The user would then have to request (through certain stringently defined entries) the projection he was entitled to via the projector. By having the relation in question only permitted to the higher ring of the projector, protection is gained by

limiting the user to accessing the relation via a well set forth sequence of code, the projector.

The problem arises of some users of GOLD STAR having ring permission as high as would the projector. This brings about a manifestation of the ring uniqueness limitation (Appendix D) of MULTICS: that a user in your ring (here, that of the projector) must be given access to a segment on an all or nothing basis; you cannot force a gate upon him.

Until such time as MULTICS allows gates to be imposed on a caller in the current ring, we must content ourselves with protecting on a projection basis only against those users in inferior rings to the system. This is the job of the projector.

The Common Usage Descriptor (CUD) (See Chapter II, Figure 2-3 for an indication how the CUD fits in the system structure).

The purpose of the CUD is to provide a description of the most common usage a relation receives. For instance, if a telephone directory were always to be accessed by "given a name, find an address and telephone" (rather than "given an address....", or "given a number...."), the CUD would somehow describe this and indicate that the ideal sorting of the directory would have name as its most ordered column.

No design or plans have yet been made for the CUD, but

it is felt that as the complexity and number of RSM's proliferates, a device like it will be needed for efficiency.

The next chapter will now explain the specifics of the implementation just described: the subroutine calls, the programming considerations, and an example of the use of GOLD STAR.

CHAPTER V

PROGRAMMING GOLD STAR

1. The User Interface

In this part of the thesis we will outline in detail the specific functional calls available from the GOLD STAR system. A few subjects are worth noting before a complete description is presented.

Arrays

GOLD STAR permits both pointers and arrays of pointers as input. This is an efficiency matter considered in a later part of this chapter, section 2.

Nesting

Since all calls in GOLD STAR are functional, calls can be nested. This means that whenever the result of an operation is appropriate, the function can be used as an argument to another call. For example, with a, b, c, d as pointers,

(a) a=mgr$union(b,c)

(b) d=mgr$union(mgr$intersect(a,b),c)

Note that all characteristics of the result will be determined by the left-most argument. Hence in (a) a will be "the same as" b (meaning if one is a quart, the other is a quart, likewise for relations; or, that the order of the CENT elements for a is the same as in b); while, with (b) d will be "the same as" a.

One other point: since PL/1 prevents us from returning arrays from a function, a pointer to an array is returned. We denote this by (p->)A where A is an array of pointers. This will affect nesting and assignment; a typical array nesting would be as follows

```
D=mgr$union(mgr$intersect(A,B)→E,C)→E
A,B,C,D,E arrays of pointers, E based
```

Notation

1. All upper case variables denote arrays
2. All lower case variables denote scalars
3. (p->)A indicates a pointer to an array
4. {} denotes a choice of elements contained
5. [] denotes what is contained is optional
6. "option" denotes a character string of one to three characters which can assume the indicated values as its characters
7. Underscored variables are character variables

Semantics

1. q & Q refer to quart pointers
2. r & R refer to relation pointers
3. All arrays in a given expression using the GOLD STAR calls must have a lower bound of one and and identical upper bounds
4. Meanings of options are:

   F = delete first argument when done (RSM)

   S = delete second argument when done (RSM)

I = insert item being referred to (DSM)

D = delete item being referred to (DSM)
(I overrides D)

1,2,....,9 = use this column of the quarts supplied as input (see get_data&successor_data) (DSM)

E = treat first argument exactly as if it were expand_quart(arg1) -- this is an efficiency feature (DSM)

5. Only one DSM or RSM can be used per operation. This means that the RSM or DSM dictated must be the same for all elements of the arrays which determine the DSM or RSM to be used.
6. Non-existent optional arguments are the same as if NULL were specified.

New Relations

Since an operation producing a relation (or a group of relations) must create new segments to hold these relations, the following procedure is followed to handle naming:

1. If F option is specified, the new relation is given the name of the one to be deleted.
2. Otherwise, the console is queried. This policy is changeable if the need arises for another approach to this case.

Error Handling

This is handled by a system program called GS_error. The user will receive a message of the form:

Error xxx Internal opcode was y

followed by two pointers, a name and pointer, or two names of interest (their meaning depends on the error code xxx) followed by an error message. Y is the internal code for the operation in progress and is indicated with each call description. Depending on the call form to GS_error, the user may be given the opportunity to return and continue from where the error occurred. The nature of the results will depend on the error incurred. Appendix G describes GS_error in greater detail. All error messages are stored in GS_err_messages with the k^{th} line being the message for error number k.

The reader is referred to the PL/1 code listings of the manager which include all of the PL/1 structures used by the system (see Appendix F), and to Appendix E, which contains the important PL/1 structures in an annotated form.

The operations are classed into six groups. They are:

1. Pure set theoretic -- those outlined in Chapter III, section 1.
2. Ordered set theoretic -- those considered in Chapter III, section 2.

3. Structural utilities -- for conversion to and from quarts and relations, and to and from scalar quarts and arrays of quarts.
4. ENT utilities -- for examination and manipulation of the ENT's.
5. Token creation and maintenance -- the creation and examination of reference numbers in data areas.
6. System Utilities -- for the creation, initiation and deletion of data areas and relations.

Finally, the reader is referred to Appendix H which describes a useful debugging aid, and to Appendix I which contains simple examples of many of the operations performed on quarts.

Pure Set Theoretic Operations

union (un=abbreviation recognized) (U=internal opcode)

q=mgr$un(q[,{q r}][,option])

r=mgr$un(r[,{q r}][,option])

(p->)Q=mgr$un(Q[,{Q R}][,option])

(p->)R=mgr$un(R[,{Q R}][,option])

options: F,S

This is the set theoretic union of the two arguments. A null second argument is interpreted as empty; i.e. it produces a copy of the first argument.

intersect (in) (I)

Forms as in union

This is the set theoretic intersection of the two arguments. A null second argument produces a quart identical to the first except with no elements.

difference (di) (D)

Forms as in union

This is the set theoretic difference of the two arguments. A null second argument produces a copy of the first argument.

cart_prod (cp) (X)

Forms as in union

This produces the cartesian product of the two arguments. The result contains a column for every column in

the inputs, with a tuple for every possible concatenation of each of the input tuples. For example,

a	b		c	d		a	b	c	d
1	11		21	31		1	11	21	31
		X			=				
2	12		22	32		1	11	22	32
						2	12	21	31
						2	12	22	32

A null second argument produces a copy of the first.

compose (co) (C)

q=mgr$co(q[,{$^{q}_{r}$}][,eqp][,option])

r=mgr$co(r[,{$^{q}_{r}$}][,eqp][,option])

(p->)Q=mgr$co(Q[,{$^{Q}_{R}$}][,EQP][,option])

(p->)R=mgr$co(R[,{$^{Q}_{R}$}][,EQP][,option])

options: F,S

The composition is performed in the following way:

1. If there is an equivalence vector (pointed to by eqp) it must be of the form

   eqp -> width name(1) name(2) .... name(width)

   where width=sum of the orders of the inputs. Bindings are then made between the names in the equivalence vector (neq's) and the CENT's in a sequential fashion. For an example, take

input 1 → a b input 2 → c d eqp → 4 w x y z

then the bindings are a-w, b-x, c-y, d-3. An "&" (ampersand) in the equivalence vector causes a CENT to be bound to itself. eqp → 4 w & y & above would produce bindings a-w, b-b, c-y, d-d. A null pointer for an equivalence vector has the same effect as if the vector were all ampersands.

2. The resulting quart (or relation) will consist of one column (conceptually, in the relation's case) for each unique, non-blank neq. For example

eqp → 4 w x y z produces w x y z

eqp → 4 w x w z produces w x z

eqp → 4 w "" w z produces w z

3. A TENT in the result is produced from a TENT from each of the inputs such that
   a. All input columns with identical bindings have an equal refno (which goes into the resulting column headed by the binding's name).
   b. When the above holds true, other columns from the result are filled from the other input columns bound uniquely to the result column's name.
   c. Columns bound to blank names are ignored. For example, given

produce the two lines Jones_33 Maple_783-8001_Ford and Jones_33 Maple_783-8001_Chevy. Had there been no listing for Jones_33 Maple in either of the directories, the result would not include him.

project (pr) (O)

q=mgr$pr(q,eqp[,option])

r=mgr$pr(r,eqp[,option])

(p->)Q-mgr$pr(Q,EQP[,option])

(p->)R=mgr$pr(R,EQP[,option])

options: F

This projects out all columns whose name in the equivalence vector is blank. The width of the vector must be equal to the order of the input, and the values in the equivalence vector replace the original CENT's in the result. For example

a	b	c		eqp	3 x y " "		x	y
1	11	21					1	11
1	11	22				yields	3	13
3	13	23						

Note that after a column is removed, any duplications in TENT's are deleted to a single entry. The ampersand may be used to indicate that the CENT for that column is to be unchanged.

Ordered Set Theoretic Operations

sort (so) (S)

Forms as in union

This sorts the first input with respect to the order of the ENT of the second input. Both inputs must contain the same ENT elements, unless the second input is null, in which case the first input is sorted without change in ENT ordering. Note that the effect of sorting according to an equivalence vector can be had by saying

```
q=mgr$so(q',mgr$me(q',eqp))
```

where eqp points to a vector containing the ENT elements of q' in the desired order (see modify_ent later on).

successor_relation (sr) (P)

```
q=mgr$sr({q r}[,q][,option])
(p->)Q=mgr$sr({Q R}[,Q][,option])
```

options. F,S

This produces the TENT in the first argument which is immediately greater than (in the lexicographic sense) the first TENT in the second input. If the second arg is null or non-existent, the first TENT in the first argument is returned. If there is no successor, a zero-length quart is returned. For example,

given first input

a	b	c
1	1	1
1	3	1

2 4 5

1 3 1

second input			produces		
a	b	c	a	b	c
2	4	5	2	5	1
a	b	c	a	b	c
1	2	1	1	3	1
a	b	c	a	b	c
1	0	0	1	1	1
a	b	c	a	b	c
3	1	1			

Structural Utilities

convert_to_q (cq) (Q)

q=mgr$cq({$^{q}_{r}$}[,option])

(p->)Q=mgr$cq({$^{Q}_{R}$}[,option])

options: F

The first argument is converted to a quart.

convert_to_r (cr) (R)

r=mgr$cr({$^{q}_{r}$},r[,option])

(p->)R=mgr$cr({$^{Q}_{R}$},R[,option])

options: F

This converts the first input to a relation with respect to the RSM used for the second input.

expand_quart (eq) (T)

Q=mgr$eq(q)

This converts a quart of length n into an array of n length 1 quarts.

squash_array (sa) (T)

q=mgr$sa(Q)

This converts an array of quarts (with identical ENT's) into a single quart. Duplications in TENT's are <u>not</u> deleted, but the result <u>is</u> sorted. Care should be exercised not to allow duplications to hinder proper operation (the duplications can be removed by projecting without removing any

columns:

q=mgr$pr(mgr$sa(Q),eqp) (with appropriate equivalence vector)

ENT Utilities

get_ent (ge) (E)

eqp=mgr$ge({$^{q}_{r}$})

EQP=mgr$ge({$^{Q}_{R}$})

This returns an equivalence vector containing the CENT portions of the input.

modify_ent (me) (M)

q=mgr$me(q,eqp)

r=mgr$me(r,eqp)

(p->)Q=mgr$me(Q,EQP)

(p->)R=mgr$me(R,EQP)

This changes the CENT portions of the input to the contents of the equivalence vector. The returned pointers are the same as the input. Ampersand (&) may be used.

clear_ent (ce) (Z)

q=mgr$ce(q)

r=mgr$ce(r)

(p->)Q=mgr$ce(Q)

(p->)R=mgr$ce(R)

This clears to NULL the PENT portions of the input and returns the input pointer.

Token Operations

At this point, it is well to point out a practical consideration which necessitates distinguishing a CENT element from a data type. It is possible that one would desire a relation such as employee-employer; but in this case, the reference numbers for employees and employers could refer to the same data type, namely personnel or name. To allow for the case where two CENT entries must be distinct, yet refer to the same data-type (and associated data area), the following convention is adopted: for all relational operations (RSM), the entire CENT is used; for all token operations (DSM) only the first four characters are used. Hence, we could have the relation name_employee-name_employer with both columns refering to the same data type "name".

By way of explaining the PENT, let us first note that when a CENT is used by a DSM to indicate a data type (by its first four characters), this character name is associated to the data area by a pointer variable. This pointer is then stored in the PENT linked to the CENT in question. Then, in future references to the quart in question, the PENT's which are non-null can be used directly without incurring the overhead of translating the first four characters of the CENT into a pointer to the data area.

A final note: in discussing the following operations, q' will refer to the special case of an order one length

one quart. So, we will be talking about the CENT and the TENT of a singleton quart.

get_refno (gr) (R)

q'=mgr$gr(d,{type/dp}[,sptr][,ent][,option])

(p->)Q'=mgr$gr(D,{type/dp}[,sptr][,ent][,option])

options: D,I

d is a pointer to a raw data item such as string or an integer. This operation takes a data item and returns a singleton quart with the item's reference number. The data area used is determined by its name, type, or by a pointer to it (which is more efficient), dp(which can be obtained via MULTICS hcs_$make_ptr routine or by the new_data_type operation outlined in group six). The resultant CENT is type unless ent is specified, in which case that is put in. If an item is not in the data area, and I is not specified, a zero-length quart is produced; if I is specified, the item is inserted, and the new reference number returned. With D option, the refno is returned despite the item's being deleted. sptr is an optional argument provided for certain DSM's where the user must indicate after which item an insertion is to be made; it is a pointer to another data item.

get_data (gd) (D)

d=mgr$gd(q[,type][,option])

(p->)D=mgr$gd(Q[,type][,option])

options: D,E,1,2,...9

This converts a reference number to a raw data item. The data area to be used is determined by type if specified; otherwise it is determined as indicated in the preface of this group from the n^{th} CENT, where n is the number option specified (1 is the default). Basically the number option permits the user to choose which column of the quart to use. The refno used is then the first in the n^{th} column.

The E option permits an entire column to be operated upon since

(p->)D=mgr$gd(q,"E")=mgr$gd(mgr$eq(q))

The E option is much more efficient, however. Finally, even if the D option is specified, the item is returned, but then deleted; references to non-existent items yield a result of NULL.

successor_data (sd) (S)

q'=mgr$sd(q[,type][,ent][,option])

(p->)Q'=mgr$sd(Q[,type][,ent][,option])

options: D,E,1,2,...9

The reference number and data area to be used are determined exactly as in get_data, using the column number option, and CENT conversion. Waht is returned is a singleton quart (with ent for its CENT if specified, else type) containing the reference number of the item which is "next" after the input refno. The order on the data is determined

by the DSM, but reference numbers are always assigned such that numerical order is kept by the order on the data. Hence, successor_data returns the next greatest refno which is included in the data area; the input refno need not be included in the data area itself. D options causes the deletion of the item whose successor was found, if there was one, not the successor itself. Finally, a null input or zero refno returns the first refno in the area, while if no successor exists (the input is greater than or equal to the last refno in the area), a zero-length quart is returned.

System Utilities

initiate_relation (ir) (Z)

r=mgr$ir(name)

This intiates the relation "name" and clears the PENT portions of the relation (this is necessary since each new process invalidates all stored pointers by virtue of the new segment numbers assigned).

new_relation (nr) (N)

r=mgr$nr(name,r)

This creates a new relation by the name "name", and assigns to it the same RSM as the second input.

new_data_type (nd) (N)

dp=mgr$nd(type,DSM)

This creates a new data type "type" (which must be a name 4 characters in length) and assigns to it the DSM named "DSM". dp is a pointer to the new data area.

kill_data_type (kd) (K)

NULL=mgr$kd({$^{q}_{type}$})

This deletes the data type "type", if that is given, or deletes the data type associated with the first ENT element of the input quart, if that is given. (See preface to group 5.)

2. Programming Considerations and Specifics

The user is referred to Appendices E and F for the system structures which serve as interfaces, and for the code of currently existent modules.

Special Functions of the Manager

The manager serves to make the rigid calling sequence enforced by PL/1 less of a problem. In the formal language, it is not possible to omit arguments, or to have a choice of the form (pointer, string, etc.) of a given argument. For example, the call to kill_data_ type could not be handled by the standard PL/1 facilities (it takes either a pointer or string). The manager overcomes this problem using a utility which obtains for the program a pointer to the arguments, but makes no interpretation (see cu_ in Multics Manual). The arguments are then treated, on a machine dependent basis, by code designed to test for argument types as well as contents. This code then permits the user to be free with his form and receive a diagnosis of trouble from GOLD STAR (rather than from MULTICS) which is more specific in its messages.

The manager also serves a number of other vital needs. It contains all of the machine dependent code (excepting area considerations, which are mentioned later). Segment and name-space management are handled by the

Arrays and Calls

In MULTICS, the call and return sequence is relatively expensive, often taking the same time as fifty or more machine instructions. To minimize this overhead, GOLD STAR is designed to deal in arrays of pointers as well as in pointers alone. By this vehicle, the user can achieve the effect of iterative calls by passing instead whole arrays. The system routines then iterate within themselves, saving the need to iterate upon the calling sequence. Hence, the entire system is designed to effectively minimize the number of calls. Hence we find that the following two calling sequences are identical in effect.

```
dcl  (A,B,C)  (n)pointer;
dcl D (n) pointer based (x);
    :
A=mgr$un(B,C)→D;
```

is the same as

```
dcl  (A,B,C)  (n)pointer;
      :
do i=1 to n;
A(i)=mgr$un(B(i),C(i));
end;
```

The second is less efficient due to call overhead.

Paging

Care must be taken that a minimum number of pages are requested in a limited period of time. Otherwise,

excessive waiting for pages will ensue. Basically, only two remedies are available: limit code as much as possible to linear flow without external calls, and localize external references to a given page. The modular construction of GOLD STAR lends itself to these restrictions; most modules have approximately linear flow, and during a given task, usually only a strategy module, a data area (or relation segment), and the free_free segment (dynamic allocations) are needed. There is virtually no intercommunication needed among the various system components other than in a simple linear transfer of control.

Frequent Trouble Spots

One of the most troublesome errors to encounter is that that of passing around erroneous pointers. The system takes care not to permit null pointers to be transmitted where they are going to cause a MULTICS error. The user must be very careful not to use uninitialized or incorrectly set pointers, as they can cause disaster.

GOLD STAR makes extensive use of allocations, and, in fact, returns all values in based allocations. The user should beware of filling up this free_free area; the system cleans up where possible, but cannot do so in some cases. For instance, when a pointer to an

array of results is returned, that array must be freed by the user to clean up properly (single returns are handled by the system).

Dynamic Storage

GOLD STAR is suitable for manipulation of large aggregations of data. At any one time, however, only a small portion of the collection need be accessible to user and system programs. In addition, GOLD STAR utilizes a certain amount of ephemeral storage in the form of QUARTS. In order to reduce the amount of storage required at any one time the free storage facilities of PL/1 are frequently invoked. Thus QUARTS and some character strings exist only as long as a user deems necessary.

In addition to those instances where ephermal data is used, data-reference number bindings are stored in a PL/1 accessible AREA. This feature, also used by RSM_Q, allows effective management of data which is dynamic in nature, without explicit storage management by the user.

```
/*    grade list contains:  "semester", "subject_number",
      "letter_grade", "student_name"    */

   compose_ptr=mgr$cart_prod(mgr$cart_prod(
               mgr$get_refno(F_ptr,"letter_grade"),
               mgr$get_refno(subj_ptr,"subject_number")),
               mgr$get_refno(term_ptr,"semester"));

/*    this forms an order 3 quart with the CENT's "letter_
      grade", "subject_number", and "semester"    */

   fi_ptr=mgr$compose(compose_ptr,fi_ptr);
   failed_ptr=mgr$project(fi_ptr,addr(names));

/*    now get addresses of the names already obtained    */

   fi_ptr=mgr$initiate_relation("student_directory");
   fi_ptr=mgr$compose(fi_ptr,failed_ptr);
   call labels (fi_ptr);
   return;
   end;
```

AUTHOR COMMENTARY -- L. ALAN KRANING

ASPECTS OF THE MODEL

The development of any large system must presume that certain trade-offs will be made during the implementation phases, but there are certain pre-implementation and pre-development considerations that have far more significant impact than the actual generation of a working system. First of all, the needs which motivated the design of the system must be considered in detail. The GOLD STAR system is suitable for administrative data management needs for a university such as M.I.T., but it is not at all obvious how well such a system would perform in other environments where the decision structure were different. Second, despite the generality of set theory, the GOLD STAR model presented in Chapter II reflects my biases, the biases of my thesis partner, and biases of those who will use the system about the nature of relatedness. An exact definition of "a is related to b" is difficult to produce without using words as nebulous as "related". What do we mean when we say that "John" is related to "Dodge"? Difficulties in this area -- especially in determining what relations are and are not properly included within a data-base -- ultimately determine the utility of our model.

The nature and degree of trade-offs in GOLD STAR revolve about some conflict between the unordered nature of sets and

the ordering of storage in computers. Retrieval of elements from sets is not well specified, but since sets and operations on them are abstract, this is not severe. However, in an environment where sets must be represented as bit strings in a totally ordered memory, we must specify an algorithm for retrieval of any item, as well as specifying its representation. Large scale associative memories would aid in representation of sets, but current technology and economics of implementation now preclude such a situation.

The quart construction saves storage space by "factoring" the data type names out of a relation. However, many to one relations such as the name-friend relation would replicate many data elements when considered as a quart. While the quart imposes ordering implicitly by storage address, it suffers the same malady of other data structures: retrieval of information must somewhere take into account an immutable order on memory. The notions of dynamic storage management and pointer data elements alleviate part of this burden, but only at execution time. The programmer must completely specify, or accept someone else's specification, for a mapping between sets and storage cells.

AUTHOR COMMENTARY -- ANDREW FILLAT

ASPECTS OF THE IMPLEMENTATION

There is a distinct dichotomy of feelings which accompany the birth and early life of GOLD STAR.

On the positive side is the most important feature of all: the framework of the system; once truly familiar to a user, it is one that reduces many exceedingly difficult data problems to amazingly short and simple algorithms. It was, in fact, hard to come up with a non-trivial example (at least from the coding standpoint).

In addition, on the positive side, is the vast open space that GOLD STAR can expand to. As special data handling needs arise, new strategy modules can be written to increase efficiency, without ever redesigning the options which use the system.

The apprehensions are attributable in great part to my inherent skepticism. There are many nagging doubts at this point of how marketable the entire MULTICS system really is. GOLD STAR, though not designed for MULTICS alone, depends upon its host for many of its nicest features and conveniences. It seems a loss to create a one-of-a-kind system, even if it becomes an integral part of the installation.

The issue of efficiency is also a nagging one. There is no way to my knowledge that one can test the design of a module under heavy loads until enough of the entire system

is available to create such a load. Hence, there is little way to elicit the performance of the system under production conditions. Although indications to date are excellent, it somehow sits uneasily upon me to be forced to design such a complex entity as GOLD STAR so empirically.

My final and I think most serious doubt is the possible "understanding gap". After conceiving, creating, and developing GOLD STAR in collaboration with the co-author, the system has become a perspective on my thinking. It is disturbing to realize how difficult it is for a detached but interested party to conceive of the system or his problem in light of the system. It is my hope that the systematic (albeit untested) approach presented through this dissertation will allow a user to become familiar with our set theoretic conception of data.

Finally, I would hope that a wide range of users would be willing to tackle the job of learning about GOLD STAR. I am convinced that the entire administration of M.I.T. could use GOLD STAR to handle virtually every part of its problems and assignments. It is my hope that these other people will see the tremendous value in our system.

APPENDIX A

A LEMMA ON THE NESTING OF PROJECTIONS

Lemma I. $\pi_A\pi_B R(x) = \pi_{A\cap B}R(x) = \pi_B\pi_A R(x)$

Proof:

if $X = \text{dom}\cup R(x)$ then by the definition of projection in (3-8)

$$\pi_B R(x) = \{f: X\cap B \to N_D \mid (\exists g)(g\in R(x) \;\&\; f\subseteq g)\}$$

Thus,

$$\pi_A\pi_B R(x) = \{h: A\cap(X\cap B)\to N_D \mid (\exists g)(g\in R(x) \;\&\; h \subseteq g)\}$$

But

$$A\cap(X\cap B) = X\cap(A\cap B) = X\cap(B\cap A)$$

So

$$\pi_A\pi_B R(x) = \pi_{A\cap B}R(x) = \pi_B\pi_A R(x)$$

Q.E.D.

APPENDIX B

ALPHABETIC ORDER AND REFERENCE NUMBERS

Data types ordered by alphabetic sequence will usually be large in size, typically of the order of 10^1 or greater. Many types will be highly active in numbers of insertions and deletions. The large size of any data type poses no significant barriers if the number of bits composing a reference number is sufficiently great; the MULTICS implementation of GOLD STAR uses a 35 bit reference number, which limits the size of any data type to $2^{34} - 1 \approx 17.8$ billion data elements. The ability to insert and delete items with impunity is more problematic. Insertion of an item requires sparseness among reference numbers if well ordering is to be maintained. Furthermore, when either inserting or deleting an item, it is of utmost importance that previous string-reference number bindings remain unaffected. Alteration of any binding necessitates a subsequent search and modification of all relational structures where the binding was assumed valid; in large data bases this process becomes prohibitively expensive if frequently performed.

One strategy which adequately serves the needs of dynamic binding is the binary tree concept outlined in Chapter 4. The illustration in Figure B-1 extends the tree discussed in Chapter 4. Inside each node above the data element

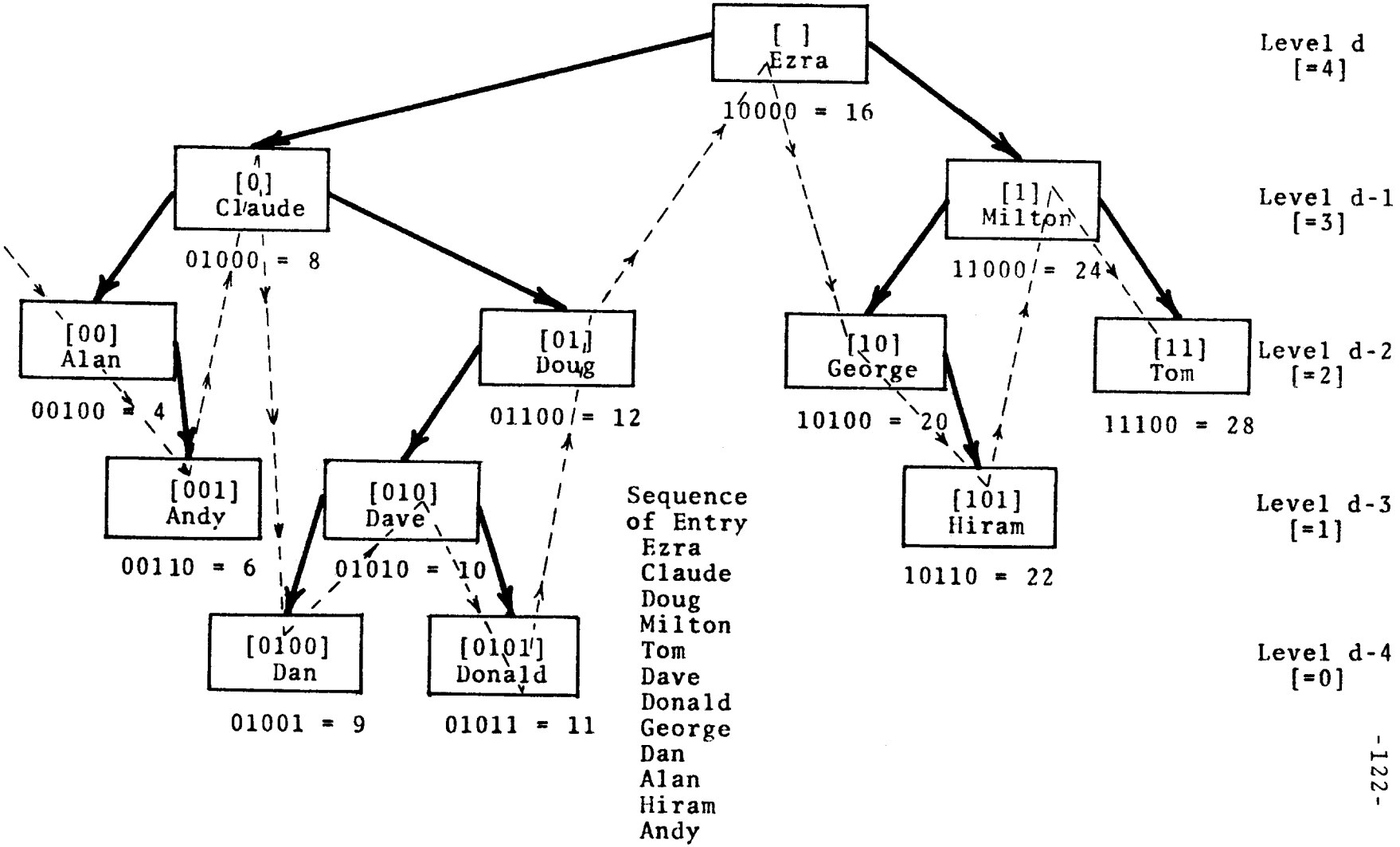
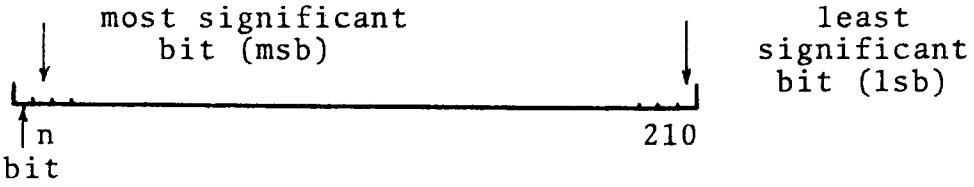


Figure B-1: The "names" Data Area

name is a bit string which, when read left to right, indicates the left and right turns required to reach an item in the tree. For example, to reach the root node, no turns are required, and to reach "Donald" turns to the left, right, left and right (0101) are called for. Two problems prevent direct usage of these "path trace" numbers as reference numbers. First, since all reference numbers are 36 bits long, the "path trace" numbers 00, 0, 000, and 0000 would be indistinguishable, as would be the set 101, 1010, 101000000, and 10100. Second, the bit strings serving as path trace numbers do not preserve the well ordering among the data elements. For example "George" lexicographically precedes "Milton", yet $path\_trace_{George} = 10$ is greater than $path\_trace_{Milton} = 1$. Uniqueness of reference numbers and preservation of order will both prevail if we adopt the following schema:

Assume that a word of storage identifies bit positions as below:



Without loss of generality, we let the i^{th} bit position $i = n-1, n-2, \ldots 3, 2, 1$ hold the flag, 0 or 1, for left and right turns respectively to be taken once we reach the i^{th} level of the tree. Note that a tree of

maximum depth n will require n bits for representation of path trace numbers.

Assume that the first 1 bit starting from the right indicates that all bits to its left are significant in a path trace. Thus the path left-left-right-left-right would result in a path trace 00101 and a reference number

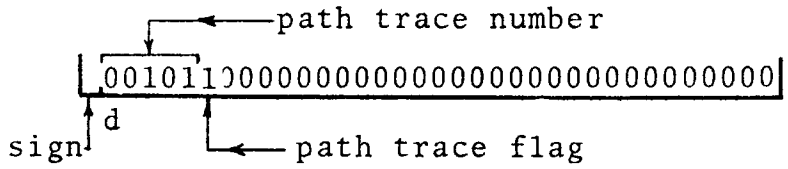


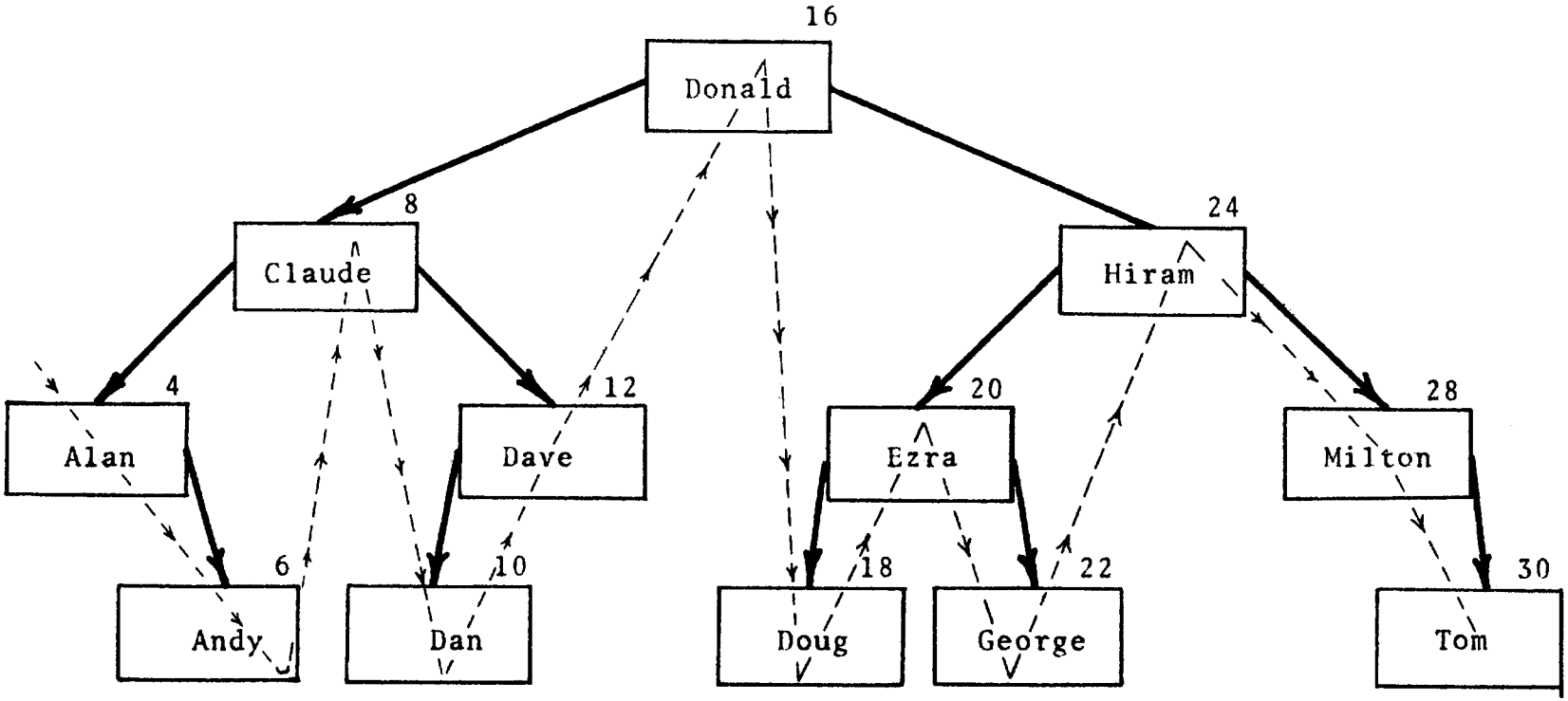
Figure B-1 shows reference numbers below each node both in binary and decial. Thus if a data item is found at depth k of a tree, its reference number can be expressed as $(\sum_{i=k}^{d} a_i 2^i) + 2^{k-1}$ with $a_i = 0$ or 1 for $i = k, k+1, \ldots d$. (2^i in binary is represented as a 1 bit in position i, if the bits are labeled as discussed above.) To compute a reference number from a string, we set i = d, and then compare the given string with the root datum. If equal, the reference number is 2^i. If the given string is less than the root, the reference number is $0 \cdot 2^i$ + the reference number obtained by 1) changing the "search" node to the root's left descendant, and 2) reducing i by 1. An analagous procedure applies to the right. If the given item is greater than the root node, then its reference number

is $1 \cdot 2^i$ + the reference number obtained by 1) changing the "search" node to the right descendant, and 2) reducing i by 1.

Notice that at each step of the search, reduction of the value of i will ensure ref#(a) < ref#(b) $\leftrightarrow$ a < b.

Overflow of the Binary Tree

Figure B-1 illustrates a tree structure resulting from one of many possible input sequences for the data type "names". Although the tree is not optimal, i.e., all nodes in which both descendants are missing lie on either level i or i + 1, fine bits are sufficient to contain all reference numbers. What happens, however, if we attempt to insert the item "Cyrus"? If we descend the tree searching for an insertion position, this item would logically be the left descendant of "Dan"; however, such an insertion cannot be made with a five bit reference number; hence we must restructure the tree before inserting "Cyrus". The Ness[A1] algorithm for restructuring trees to optimum depth yields the form in Figure B-2. The restructuring process destroys the entire set of old bindings; to retain the viability of relations and quarts in which the original bindings participated, a quart is produced, relating new and old reference numbers for each member of the data type. At restructure time, the user is informed of tree overflow, and the existence of the new-old-refno quart. He then performs composi-



Sequence of Entry:

Donald
Claude
Hiram
Alan
Dave
Ezra
Milton
Andy
Dan
Doug
George
Tom

Quart for Restructure of Relations Involving "names" data type-

old	new
4	4
6	6
8	8
9	10
10	12
11	16
12	18
16	20
20	22
22	24
24	28
28	30

Figure B-2: Restructured "names' Data Area

tion of this quart with the necessary relations. The tree overflow problem will in practice arise very infrequently; however when overflow does occur, it is mandatory that recovery procedures by operable.

Randomizing The Input Sequence for dsm_astring

Referring to Figures B-1 and B-2 we note that the particular organization of that tree depends to some extent upon the sequence in which data elements are entered. In particular, if we insert a large number of data elements which are input in alphabetic order, then the tree will quickly overflow. Alleviation of this problem is accomplished by scrambling large sets of input data prior to the binding process.

APPENDIX C

A FORMAL SYNTAX FOR A PARSER

<Regular Expression>=::<EXP1>/<NUMERIC EXP><VARIABLE>

<EXP1>=::<EXP2>|<EXP2><OR><EXP1>

<EXP2>=::<EXP3>|<EXP3><EXP2>

<EXP3>=::<PEXP4>|<PEXP4>*

<PEXP4>=::"<literal>"|$|?|!|.|<|>|^<PEXP4>|
(<Regular Expression>)

<NUMERIC EXP>=::<NUMERIC EXP1>|<NUMERIC EXP1><OR>
<NUMERIC EXP>

<NUMERIC EXP1>=::<NUMERIC EXP2>|<NUMERIC EXP2><NUMERIC EXP1>

<NUMERIC EXP2>=::<NUMERIC EXP3>|<NUMERIC EXP3>+|
<NUMERIC EXP3>-

<NUMERIC EXP3>=::<digit strings>

<VARIABLE>=::<character strings>

definitions: <OR>≡|
$ ≡ any digit
? ≡ any upper case character
! ≡ any lower case character
. ≡ anything
< ≡ left anchor
> ≡ right anchor

Note: any special character appearing twice in succession is interpreted as one of itself.

APPENDIX D

PROTECTION ON MULTICS

1. The Basic Concepts

Each segment in MULTICS possesses an access control list (ACL) which lists all processes which may access that segment, and the restrictions on that access. There also exists a common access control list for each process (CACL) which can be viewed as simply an ACL common to all of a user's segments. The possible forms of access are read (R), write (W), execute (E), and append (A), or no access (this appears as a blank in the ACL but will be signified by either - or X). It should be noted that by process, what is meant is a thread of control; i.e., no matter who the segments "belong to", a series of calls, returns, and execution represents the thread of control of a process. It is important to note the ACL discriminates by this process concept.

As well as an REWA attribute, each process listed in the ACL possesses a ring bracket, of the form a:b:c. This ring bracket specifies the action the supervisor will take upon a call from this listed process depending on which ring the calling process is in. The general concept of rings is merely an extension of the two state system to 64 states; operations permissible (in this case, by operations we mean access and calls) grow progressively more restricted as we move out from ring 0 to ring 63. Part 3 is a discussion of how rings work, and the following diagram should indicate how rings serve an

analogous purpose to states.

Ring bracket listed in called segment for calling process	a :	b :	c	
Calling process in ring	0-(a-1)	a-b	(b+1)-c	(c+1)-63
Effect of a call:	Legal New Ring=a	Legal Ring Unchanged	Legal if call is to "gate" New Ring=b	Illegal
Effect of non-call access:	Illegal	Legal	Illegal	Illegal

This is all subject to the REWA permission listed for the calling process. In words, the essence of this ring bracket is as follows: if the calling segment is in the first specified range (a to b), then all forms of access are legal, and no ring changes occur; if the calling segment is in the second range (b+1 to c), only a call is permitted, and this generates a "gate crossing fault" (the fault is analogous to the exception generated when a problem state program makes a call on the supervisor; entry to the more privileged routine must be at specified places, or "gates"); if the calling segment is in the low outside range (less than a), only a call is permitted, and an "attempt to execute data fault" is generated (the name is misleading as Part 3 indicates, but the situation is analogous to a supervisor calling a problem program-care must be taken to allow the less privileged called program only the arguments passed to it, and not to allow any access to the calling programs' more privileged data); if the

calling segment is in the high outside range (greater than c), no access is allowed.

This gate scheme has distinct advantages over a simple two state system. First, by using ring 0 for the supervisor, the HCS can be fully protected from undesired access by specifying a ring bracket of 0:0:i (i any other ring) for all processes. Yet, under this view, the HCS becomes merely a copy of code available to everyone; system processes (usually called daemons) can run in any ring, and still have full access to the HCS if its process is permitted from any ring, without allowing other users unrestricted access. The second major advantage is almost a corollary to the first; by writing a sub-system in ring i, any user can possess supervisor type privileges over his system by having his users in ring i+1. The logical consistency of this plan is clear: the HCS is to its sub-systems as the sub-system is to its users. Since each user can itself be a sub-system, the uniformity of this scheme is fantastic. The ability to assign access on a per process basis, combined with the refined and extended system of states -- the rings -- leads to a computer system which is virtually tailored to the sub-system writer as well as the average, albeit necessarily skilled, user.

There is one major limitation upon the flexibility of the ring system. I shall call this the "ring uniqueness limitation". The name describes the fact that if a user has privilege to some ring, then there is no way that another process

with the same ring privilege can permit access to the first process on a subordinate basis. For example, if Jones has a sub-system in ring 3 (his users run in ring 4), and Smith has ring 3 permission, then Jones must make the all-or-nothing choice of whether to give Smith unrestricted access or no access at all. This situation is clearly one that does arise on any large system like MULTICS, where there would be much demand for equal and independent sub-system space.

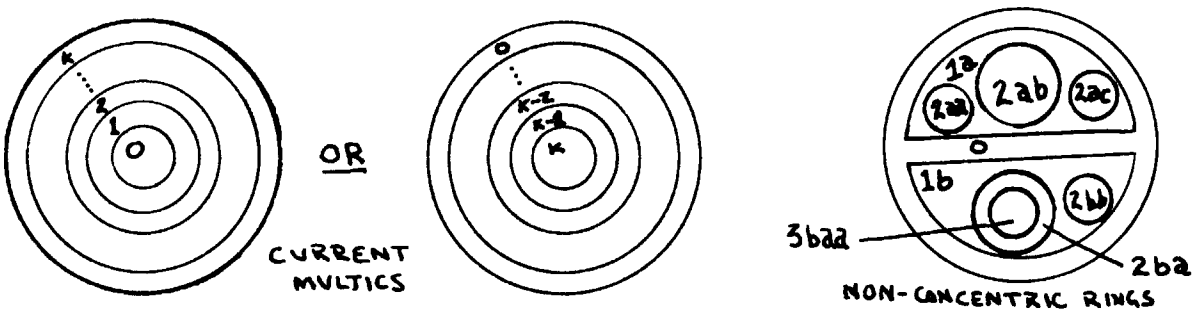
2. Eliminating The Ring Uniqueness Limitation

Eliminating the aforementioned limitation is clearly a useful and advantageous goal to adopt, at least from a theoretical aspect. What would result would be a totally controllable environment of sharing, where the level of privacy could range from zero to total. There is, however, a serious question of whether any scheme could both remove the restriction and not cost dearly in overhead.

Three different schemes are presented which, through different approaches, would eliminate the MULTICS restriction. These ideas are the creation of non-concentric rings, the use of procedure as well as process access control, and the effecting of a trap mode of access. It is not possible for the author to make detailed analysis of the implementations or the overheads, for simple lack of facts; however, general assessments based on a knowledge of the system overview and the ring system will be made.

A. Non-concentric Rings

The idea of non-concentric rings is a reasonably simple conceptual extension of the current ring system. A diagramatic presentation would be the best introduction.

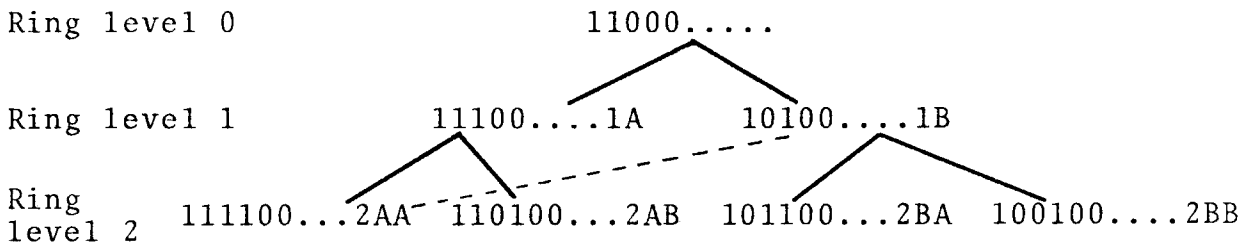


The non-concentric rings (NCR's) would effectively define what might be called a "sub-system context". The advantageous feature of this system would be that a user might have ring 3 permission in his own sub-system context, and have only ring 5 permission in another such context. The NCR's would isolate sub-systems into their own branch of the tree, and would even define independent scopes of authority on the higher level rings. The NCR's permit allotment of sufficient privilege without the allocation of over-sufficient permission.

A major problem would certainly be the excessive space required to store information about ring status. In its full extension, NCR's would have a theoretically infinite number of subrings at every level. A practical solution, which is not especially costly in a theoretical sense, is to have a binary tree structure for the ring system. There would be one ring 0, two ring 1's, four ring 2's, and so on. Under

this plan, a process' place on the ring tree could be determined using n+1 bits, where n is the number of levels in the tree (the one extra bit would be used as a flag to mark the end of significance in the left to right scan of the binary bit string). The following diagram indicates the binary format employed. Note that the binary method places a restriction on the operations staff: sub-systems must be placed on a low enough level to permit space enough for all the co-equal systems. In practice, ring 6, 7, or 8 would be the likely location for the average sub-system (thus permitting 64, 128, or 256 sub-systems).

Tree Structure of Rings



There is another major problem involved with the use of the NCR's. That is, how do we determine faults? Clearly, there are two types of faults (disregarding the distinction between upward and downward): the standard type -- as from 2AA to 1A -- and a tree-crossing fault -- as from 2AA to 1B, the dotted line. The standard upward and downward faults are easily determined by being on connected branches -- simple up or down movement on the tree -- and would be handled exactly as in MULTICS. The tree-crossing fault presents a problem:

if we allow faulted access to gates across the tree, we run into tremendous problems specifying what corresponds to a ring bracket. A variable length list would be needed for each entry in the ACL specifying from which branches of the tree a process could call from. Another problem: how does the system distinguish between the faults?

A possible solution to these problems might be to not allow cross-tree access at all. Then, the system could search for rings by simply following the branches of the tree. A cross-tree access would be detected if the system has to change direction in its tree search; i.e., if the tree search required going up and then down branches. Ring brackets would remain unchanged, and would only refer to the branches of the tree reachable without a cross-tree fault arising. For example, a segment would be specified with brackets like 1A:2:3; this would permit access unrestricted from 1A,2AA, and 2AB, and faulted access from rings 3AAA, 3AAB, 3ABA, and 3ABB.

But note that this solution requires one important addition to the features allowed a process: a process would have to be able to change its current ring so as to be able to reach all rings to which it is permitted. That is, a user with ring 1A permission who wishes to use a system in ring 1B must be able to change his ring to the 2BA he has been permitted to by the 1B system writer. This ring changing facility would require changes to the supervisor which would maintain a list of permissible rings for each process (currently, this

list has only one entry). Facility for users to grant appropriate permissions would also have to be added, with appropriate safeguards, such as another list for acceptable ranges into which permission might be granted.

The system of non-concentric rings is conceptually the most rigorous solution to the ring uniqueness limitation of MULTICS. In its ideal form, with cross-tree faults being treated analogously to regular faults by use of an expanded ACL, the system should be easy to use, albeit difficult to implement. In the modified form, with cross-tree reference allowed only by a conscious change of rings, use would be more complicated, but implementation possibly easier. In any form, this system would be difficult to implement, and might not prove the best way to overcome the restriction of ring uniqueness despite its conceptual advantage.

B. Procedure and Process Oriented Access Control

The principle involved with this method of overcoming the ring uniqueness limitation is a very simple one. As well as specifying what processes may access a given segment, a user could also specify from what procedures (segments) these processes could issue the calls. A sample version of the current and extended ACL's might show how such a specification would appear.

Current Form

Multics.Jones.* RE 1:1:8

Admin.Smith REWA 1:6:63

Extended Form

Multics.Jones.* System.Andy RE 1:1:8

Admin.Smith Exec.* REWA 1:6:63

What this form of extension does is to specify from what procedure a process must make a call to be permitted. In the example, only if Smith of the Admin project were calling from a segment named Exec.something would he be allowed access.

The above scheme does circumvent the ring uniqueness restriction by allowing a sub-system writer to specify that his segments are callable only by other segments of his system. This would then prohibit other users, even those with high enough ring permission, from accessing critical segments on their own. As long as the dispatching segments which could be referred to by outside programs were permitted only to execute (and not to be modified), the integrity of the calling sequences could be maintained.

There clearly are limitations on this type of system. Space would certainly prove some sort of factor where it was necessary to specify complex lists of permitted callers. Some degree of overhead would certainly be introduced in checking the validity of the calling procedure. This method would go to pot if write permission were involved at any level. But a more fundamental problem might be the maintenance of the integrity of pathnames. Although pathnames are

unique, there seems to be no assurance that a clever programmer could not pass to a called program an erroneous pathname. If the system were to completely handle identification of calling procedures, the overhead involved in verification might prove unworkable.

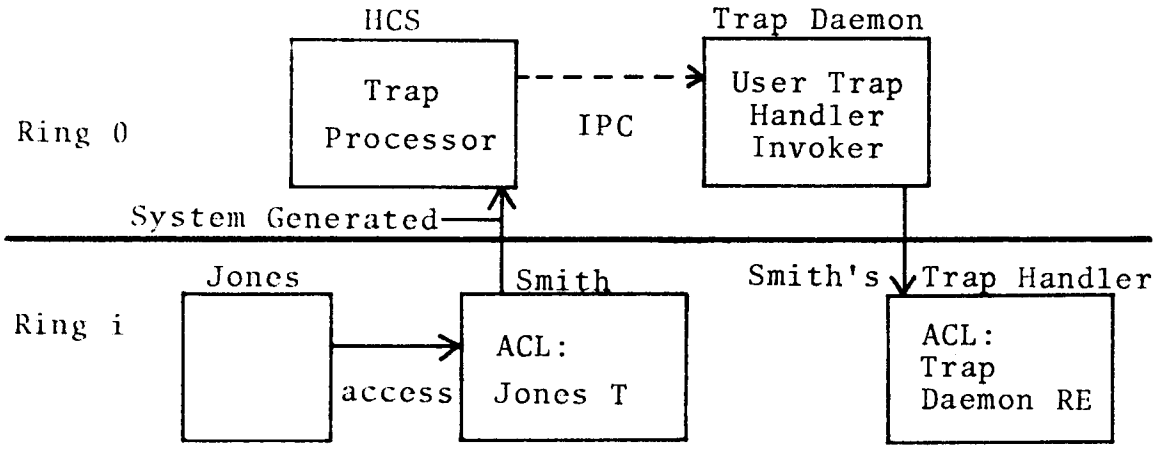
Though this plan has merit as a conceptually simple one, it would be a hard plan to carry out effectively. Safeguards would be required against tampering with name conventions, and proper permission specifications by users would become more crucial and difficult to keep track of. This scheme might, however, be the easiest to implement, at least relative to others.

C. Trap Mode

From an efficiency of operation standpoint, the effecting of a trap mode of access would be the best solution to the ring uniqueness problem. A specification of trap mode for a calling process would automatically invoke a trap processing program upon issuance of a call. This trap processor could do anything, including freeing the caller from future traps on the same segment. In the case of ring uniqueness, a sub-system writer could assure that the caller was in a properly low ring by specifying trap for all calls from rings too high (low in number). The advantage of this method is obvious: special action is taken only on invalid reference attempts; valid access will generate a zero overhead.

The problems in implementation are considerable. They would include such items as maintaining the integrity of the trap procedure itself (note that if the trap procedure is run under the calling process, the caller has permission to execute, and might find it useful to run the trap procedure independent of any real trap). Disabling the trap for a fixed period might also be difficult to accomplish without loss of efficiency. In fact, though the original plans for MULTICS included provisions for a trap mode of access, complications in the design forced its exclusion from the current system.

A proposal for the inclusion of trap mode is outlined below. To the author's knowledge, this plan is feasible, although shortcomings in the interprocess communication facility would make a completely tight, safe system difficult. This shortcoming will be mentioned and considered later. A diagram would illustrate the basic operation of the system. The name appearing above each segment indicates its owner.



mine who had desired its activation. Argument passing presents a related difficulty to the system.

The only possible solution under the current system design is to employ an array of "mailboxes". These would be segments possessed by the trap daemon, and allowed on a one per user basis to other processes. The daemon could then determine who had issued the IPC by checking the mailboxes; he could be assured that the proper process has issued the request because only one process is permitted per mailbox. The two potential difficulties that arise are space requirements, and the possible confusion of sequence brought about by the asynchronous nature of the IPC requests.

An implementation of trap mode, though not allowing as complete a flexibility as would non-concentric rings, would certainly provide that extra leaway of freedom to expand upon the available sub-system capabilities. The ring uniqueness limitation would be eliminated in an indirect way. Those programs which require special screening due to ring permission would experience considerable overhead, but those permitted only to low rings (high in number) would be unaffected by the safeguards imposed through trap mode. Trap mode is certainly not an easy concept to implement, but it might well prove a worthwhile investment in time and effort.

Eliminating the single big restriction of the MULTICS protection method is to be a virtual necessity as the size

and applicability of the system increases, and more subsystems come under its purview. Which of the three methods outlined is taken, or whether another method is used is not so crucial. It is most likely that overheads, and not theoretical advantage, will determine the final choice. It is hard to divorce oneself from the idea that a free sharing environment is the ideal, given the ability to totally regulate its scope. It is this idea of a full range -- from total privacy to total sharing -- that must be the ultimate goal of protection design.

Just as generalization brings flexibility, it also brings overheads and time consumed where the activity might have been foregone. It is the ultimate futures of MULTICS-like systems which will bring at least a statistical answer to the trade-off at hand.

3. How Rings are Implemented in MULTICS

Associated with each process is a set of descriptor segments, one for each ring in which the process has access defined in the ACL by ring brackets. A schematic representation would prove useful: (x=no access; i=any ring of higher number)

Segment name	DS 0	DS 1	DS 2	DS 3	DS 4 ...DS	I Bracket in ACL
aaaa	R	RE	x	x	x	1:1:i
bbbb	R	R	RE	x	x	2:2:i
cccc	R	RE	RE	RE	x	1:3:i

The ring of current execution is determined by which descriptor segment (DS) the descriptor base register points to. The following enumeration of cases describes how the file system uses rings.

A. cccc calls or refers to bbbb; current ring is 2

All attempts at access are legal, as governed by the RE permission.

B. bbbb calls aaaa; current ring is 2

1. no access is found in the ring 2 DS; a gate crossing fault causes the gatekeeper to obtain control (the gatekeeper is conceptually an independent overlord)
2. gatekeeper examines DS 0-1 for E access in aaaa
3. if access is found, first ring in (highest number) is used; third element of bracket is checked to assure call is from a valid ring
4. referenced entry point is validated as a gate
5. if all is OK, DBR is loaded to point to DS 1

C. bbbb refers to aaaa; current ring is 2

Non-call references are illegal; the effect is the same as accessing another segment with no permission at all.

D. aaaa refers to bbbb; current ring is 1

Read is permitted; WA is permitted only if the ring bracket of bbbb includes the current ring in the first range (a:b).

E. aaaa calls bbbb; current ring is 1

1. since there is read permission, an attempt to execute data fault is generated, and the gatekeeper takes control
2. gatekeeper checks DS 2-63 for E access
3. first ring down with E is used; no gate is checked for
4. DBR is reloaded to DS 2; appropriate arguments are made available to the lower ring bbbb

There is one safeguard worth noting. The system will not allow a ring bracket to be assigned to any segment such that any element of the bracket specifies a ring lower in number than the ring it is being created from (the current ring). Thus, a user in ring 2 cannot assign 1:1:3. This guards against a user writing a lower number ring routine, and then transferring to it, thereby giving him higher level privilege than he should have.

One other note: a called segment will run in the same ring as the caller is in, unless a fault is generated; in that latter case, the new ring will be the nearest ring numerically to the calling ring.

APPENDIX E

SYSTEM INTERFACES

1. User to System Interface

The option writer or user of GOLD STAR needs basically the PL/1 environment and temporary storage facilities, and a collection of segments which he wishes to use with the system, and which were created by it. He need have no knowledge of the segments except their names, and the data types they contain if they are relations. In the case of DSM's, the segment name is the data-type. With RSM's, it is suggested that the segment name reflect the data-types related (e.g., the relation segment "name_addr" should relate data types "name" and "addr".

The user deals with GOLD STAR exclusively with pointers, segment names, and strings (see section V-1). There is a very small group of items to which these pointers refer: quarts, relations, equivalence vectors, "data", and data areas. The structures are described as follows:

```
declare 1 quart based (quart_ptr)
  2 ID fixed binary (35) initial (0),
  2 length fixed binary (35),
  2 order fixed binary (35),
  2 CENT (allocation_order refer (order) character (32),
  2 PENT (allocation_order refer (order) pointer,
  2 TENT (allocation_length refer (length),
```

```
            allocation_order refer (order)) fixed binary (35);
    declare 1 equivalence_vector based (equiv_ptr),
      2 width fixed binary (35),
      2 element (allocation_width refer (width)) character(32);
    declare 1 data_item based (data_ptr),
      2 length fixed binary (35),
      2 string character (allocation_length refer (length));
```

Note that the data item definition is merely a way of describing a based varying character string. The relations and data areas are PL/1 areas with the first allocation containing as its first item the character (16) RSM or DSM name respectively.

All other names and character strings are assumed to be fixed strings of whatever length is called for, if any.

2. Internal Interface

The manager translates the rather free form of arguments it receives into one of two structures for RSM's and DSM's. These argument structures are the sole communication between the system modules.

The following is an annotated version of the PL/1 structures used:

```
    declare 1 dsm_arguments based (arg_ptr),
      2 upper_bound fixed bin(17), This tells how many ele-
                                   ments were in input arrays
      2 column_index fixed bin(17), This is the 1, 2, ... 9
                                   option
```

```
   2 operation_code character(1), This tells which oper-
                                  ation is to be performed
   2 d_or_i_option character(1), This contains either
                                  "D" or "I" if called for
   2 e_option character(1), This contains "E" if called
                                  for
   2 data_area_ptr pointer, This is filled in by the
                                  manager from the appro-
                                  priate given information
                                  (CENT, PENT, or given
                                  name)
   2 data_or_quart_ptr (limit refer (upper_bound))
   pointer, These are the first arguments
   2 return_ptr (limit refer (upper_bound)) pointer,
                                  These are for the re-
                                  sults.  If sptr's were
                                  provided in a get_refno
                                  operation, they are put
                                  in here.

declare 1 rsm_arguments based (arg_ptr)
   2 upper_bound fixed bin(35), This gives the upper_
                                  bound of the input arrays
   2 operation_code character(1), This tells the opera-
                                  tion to be performed
   2 quart 1_ptr (limit refer (upper_bound)) pointer,
```

2 relation_1_ptr (limit refer (upper_bound)) pointer,
2 quart_2_ptr (limit refer (upper_bound)) pointer,
2 relation_2_ptr (limit refer (upper_bound)) pointer,
These are the inputs. The manager determines whether they are quarts or relations and places their pointer in the appropriate slot. Non-existent arguments are null.
2 equivalence_vector (limit refer (upper_bound)) pointer, These are the equivalence vectors for compose. Other operations needing eqv place the pointer in quart 2_ptr.
2 return_ptr (limit refer (upper_bound)) pointer,
These are for the results. They are null if the result is to be a quart, or point to a new segment of a relation is to result. It is the responsibility of the RSM to initialize the new segment properly.

APPENDIX F

CURRENT SYSTEM CODE

Preface

The following modules are included in PL/1 source form:

mgr_declare -- these are the declarations for the manager

mro -- this is the portion of the manager which handles calls to union, intersect, difference, compose, project, cart_prod, sort, successor_rel, convert_to_r, convert_to_q, modify_ent, get_ent, and clear_ent.

oro -- this is the portion of the manager which handles calls to new_relation, initiate_relation, and squash_array.

ero -- this is the portion of the manager which handles calls to expand_quart.

mdo -- this is the portion of the manager which handles calls to get_refno, get_data, and successor_data.

odo -- this is the portion of the manager which handles calls to new_data_type and kill_data_type.

GS_error -- the error handler (see Appendix G).

GS_err_messages -- the error messages (see Appendix G).

gsdb -- the system utility program (see Appendix H).

dsm_astring -- this is the DSM which handles alphabetic strings to be placed in alphabetic order (see Appendix B).

dsm_integer -- this is the DSM for integers.

dsm_chain -- this is the DSM for alphabetic strings to be placed in a non-alphabetic order.

dsm_table -- this is the DSM for alphabetic strings which are not in alphabetic order, and which are involved in frequent retrieval, and infrequent updating.

rsm_wq -- this is the RSM which performs all operations (except, of course, convert_to_r) on two quarts.

rsm_q -- this is the RSM which handles quarts to be kept permanently. RSM_Q calls RSM_WQ to operate on the quarts, and only handles storage and retrieval of quarts to and from auxillary segments.

tree -- this is the algorithm to restructure a tree of a dsm_astring data area. TREE produces an optimal tree, and a quart of new and old reference numbers to use to update all relations using that data type.

mgr_declare.incl.pl1 05/20/70 1116.4 edt Wed

```
          declare 1 parameter_structure based (param_ptr),
              2 arg_no bit(17) ,
              2 filler bit(2),
              2 display_if_8 bit(17),
              2 desc_no bit(17),
              2 multics_junk bit(19),
              2 a_ptrs (100) pointer; /*pointers are compute
d by formula*/
/*The following is the technically correct version of the pa
rameter structure.
   Note that the display pointer exists only if the di8 half
 word is 8. The actual
   method used is to compute an offset to use, since this is
 faster than this
   adjustable structure method.
          declare 1 arg_structure based,
              2 arg_no bit(17),
              2 filler bit(2),
              2 di8 bit(14),    use of bit 14 has effect of
dividing by 8
              2 filler bit(3),
              2 desc_no bit(17),
              2 filler bit(19),
              2 arg_ptrs (arg_no) pointer,
              2 display_ptr (di8) pointer,
              2 desc_ptrs (desc_no) pointer;
                    */
          declare 1 descriptor_structure based (desc_ptr),
              2 dt_code bit (15) ,    /*data type code - see
 MULTICS implementation of PL/1*/
              2 other_stuff bit(3),   /*not used*/
              2 string_size bit (18),
              2 low_bound fixed bin(35) ,
              2 top_bound fixed bin (35);
          declare 1 quart  based (gen_purpose_ptr),
              2 ID fixed bin(35) initial(0),   /*zero ID mea
ns quart*/
              2 qlength fixed bin(35),   /*not used here*/
              2 order fixed bin(35),    /*not used here*/
              2 cent(alloc_order refer (order)) char(32) ,
              2 pent(alloc_order refer (order)) pointer,
              2 tent(alloc_length refer (qlength),alloc_orde
r refer (order)) fixed bin(35);
```

```
          declare 1 arg_array_rsm  based (arg_ptr),
              2 upper_bound fixed bin(35) ,
              2 opcode character (1),
              2 q1_ptr (limit refer (upper_bound)) pointer,
              2 r1_ptr (limit refer (upper_bound)) pointer,
              2 q2_ptr (limit refer (upper_bound)) pointer,
              2 r2_ptr (limit refer (upper_bound)) pointer,
              2 equiv_ptr (limit refer (upper_bound)) pointe
r,
              2 return_arg (limit refer (upper_bound)) point
er;
          declare 1 arg_array_dsm  based (arg_ptr),
              2 high_bound fixed bin(17),
              2 column_index fixed bin(17) initial(1),
              2 opcode character(1),
              2 d_or_i_option character(1),
              2 e_option character(1),
              2 data_data_ptr pointer,
              2 d_or_q_ptr(limit refer (high_bound)) pointer

              2 return_arg (limit refer (high_bound)) pointe
r;


          declare fixed_plate fixed bin(35)  based (gen_purp
ose_ptr);  /*fixed bin template for quart.ID*/
          declare oset bit(18) based (gen_purpose_ptr); /*Th
is is for the area manipulation*/
          declare label_plate char(16)  based (gen_purpose_p
tr);   /*this is char(16) template for ID slot*/
          declare return_structure (limit) pointer  based (r
eturnable_ptr);  /*returned to the caller*/
          declare option character (oplength)  based (op_ptr
);
          declare based_nam character (4)  based (gen_purpos
e_ptr);
          declare rel_name character (lentty)  based (gen_pu
rpose_ptr);
          declare based_arg character(as_len) based (gen_pur
pose_ptr);
          declare p pointer  based (gen_purpose_ptr);
          declare indexor (kk) pointer  based (gen_purpose_p
tr);  /*this acheives the kk th pointer*/


          declare callname char(16) aligned;
          declare nam character(4) aligned;
          declare options character (3) aligned;
          declare ent character (32) aligned;
          declare op character(1) aligned;
          declare wdir character(168) aligned;
```

```
          declare unique_chars_ entry external returns (char
acter (15));
          declare (error_table_$noarg,error_table_$segknown,
error_table_$name_not_found)
              fixed bin(17) external;
          declare (xfer,other_ptr,arg_ptr,desc_ptr,gen_purpo
se_ptr,op_ptr,param_ptr,returnable_ptr) pointer;


          declare access_code fixed binary(5) initial(01011b
);
          declare (kk,kik,jj,mm,iii,limit,dlen,oplength,dose
t,lwdir) fixed bin(17);
          declare (lentty,err_code,new_place,alloc_order,all
oc_length) fixed bin (17);
          declare (two_arg_sw,three_arg_sw,arg_array_sw,skip
_3_sw) fixed bin(1) initial (0b);
          declare (option_sw,nam_sw,ent_sw,ptr_sw,squash_sw,
F_sw,S_sw) fixed bin(1) initial(0b);
          declare (as_len,op_bump) fixed bin(17);
```

mro.pl1 05/20/70 1011.0 edt Wed

```
mro: procedure returns(pointer);

          %include mgr_declare;

un:union:entry returns(pointer);
          op="U";
          go to setup1;
in:intersect: entry returns(pointer);
          op="I";
          go to setup1;
di:difference: entry returns(pointer);
          op="D";
          go to setup1;
cp:cart_prod: entry returns(pointer);
          op="X";
          go to setup1;
so:sort:  entry returns(pointer);
          op="S";
          go to setup1;
sr:successor_rel: entry returns(pointer);
          op="P";
          go to setup1;
cr:convert_to_r: entry returns(pointer);
          op="R";
          go to setup1;
cq:convert_to_q: entry returns(pointer);
          op="Q";
          go to setup1;
co:compose: entry returns(pointer);
          op="C";
          go to setup1;
me:modify_ent: entry returns(pointer);
          op="M";
          go to setup1;
ge:get_ent: entry returns(pointer);
          op="E";
          go to setup1;
ce:clear_ent: entry returns(pointer);
          op="Z";
          go to setup1;
pr:project: entry returns(pointer);
          op="O";
          go to setup1;
```

```
setup1: /*Here we do argument processing*/
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no-1;
          else doset=arg_no;
          if arg_no<2 then call GS_error$p(12,op,null,null);

/*Error here is a null argument call in*/
          if a_ptrs(1+doset)->dt_code=29 then do;
              /*We have first arg is array of pointers*/
              arg_array_sw=1b;
              limit=a_ptrs(1+doset)->top_bound;
              if a_ptrs(1+doset)->low_bound¬=1 then call GS_
error$p(14,op,a_ptrs(1),null);
/*Error here is non one lower bound*/
          end;
          else if a_ptrs(1+doset)->dt_code=13 then do;
              arg_array_sw=0b;
              limit=1;
          end;
          else call GS_error$p(13,op,null,null);
/*Error here is failure to have pointer or array of pointers
 for first arguments*/


          op_bump=a_ptrs(arg_no-1+doset)->dt_code;
          if op_bump=520|op_bump=522 then do;
              call cu_$arg_ptr(arg_no-1,op_ptr,oplength,err_
code);
              if index(option,"F")¬=0|index(option,"f")¬=0 t
hen F_sw=1b;
              if index(option,"S")¬=0|index(option,"s")¬=0 t
hen S_sw=1b;
              op_bump=1;
          end;
          else op_bump=0;
          if op="C" then kik=0;
          else kik=1;
          if arg_no-1-op_bump>3-kik then call GS_error$p(15,
op,a_ptrs(1)->p,null);
/*Error here is too many arguments*/
          do kik=2 to arg_no-1-op_bump;
              if limit=1 then do;
                  if a_ptrs(kik+doset)->dt_code¬=13 then cal
l GS_error$p(16,op,a_ptrs(1)->p,
                      a_ptrs(2)->p);
/*Error here is inconsistant upper bound*/
              end;
              else do;
                  other_ptr=a_ptrs(kik+doset);
                  if other_ptr->dt_code¬=29 then call GS_err
#or$p(17,op,a_ptrs(1)->p,null);
/*Error here is inconsistant upper bound or type*/
```

```
                    else if other_ptr->top_bound¬=limit then c
all GS_error$p(18,op,
                        a_ptrs(1)->p,a_ptrs(kik)->p);
/*Error here is inconsistant upper bound*/
                    else if other_ptr->low_bound¬=1 then call
GS_error$p(14,op,a_ptrs(1)->p,null);
/*Error here is non one lower bound*/
                end;
                if kik=2 then two_arg_sw=1b;
                else three_arg_sw=1b;
            end;


    /*Here we compare argument structures with those permiss
ible by operation*/
            if two_arg_sw & (op="Q"|op="E"|op="Z") then  call
GS_error$p(19,op,a_ptrs(1)->p,null);
/*Error here is two arguments with cq,ge, or ce*/
            if ¬two_arg_sw & (op="M"|op="O"|op="R") then call
GS_error$p(20,op,a_ptrs(1)->p,null);
/*Error here is omission of mandatory second arg in cr or me
*/
            if F_sw & (op="E"|op="M"|op="Z") then F_sw=0b;
            if S_sw & (op="Q"|op="R"|op="E"|op="M"|op="Z"|op="
O") then S_sw=0b;


    /*Here allocate argument array structure and go to work*
/
            allocate arg_array_rsm;


setup2:     do kk=1 to limit;
                if op="C"  then do;
                    if three_arg_sw then arg_array_rsm.equiv_p
tr(kk)=
                        a_ptrs(3)->indexor(kk);
                    else arg_array_rsm.equiv_ptr(kk)=null;
                end;
                if ¬two_arg_sw|a_ptrs(2)->indexor(kk)=null the
n do;
                    arg_array_rsm.q2_ptr(kk),arg_array_rsm.r2_
ptr(kk)=null;
                    if op="R" then go to err_20;
                end;
                else if a_ptrs(2)->indexor(kk)->fixed_plate=0|
op="M"|op="O" then do;
                    arg_array_rsm.q2_ptr(kk)=a_ptrs(2)->indexo
r(kk);
                    arg_array_rsm.r2_ptr(kk)=null;
                    if op="R" then do;
err_20:                 free arg_array_rsm;
                        call GS_error$p(20,op,a_ptrs(1)->p,a_p
trs(2)->p);
```

```
                end;
            end;
            else do;
                arg_array_rsm.r2_ptr(kk)=a_ptrs(2)->indexo
r(kk);
                arg_array_rsm.q2_ptr(kk)=null;
                if op="P" then do;/*Error handling$$$*/
                    free arg_array_rsm;
                    call GS_error$p(21,op,a_ptrs(1)->index
or(kk),r2_ptr(kk));
                end;
/*Error here is attempt to get successor of a relation*/
            end;


    /*First operand is examined here*/
            if a_ptrs(1)->indexor(kk)=null then do;/*Error
 handling$$$*/
                free arg_array_rsm;
                call GS_error$p(22,op,null,null);
            end;
/*Error here is a null first operand*/
            if a_ptrs(1)->indexor(kk)->fixed_plate¬=0 then
 do;
                arg_array_rsm.q1_ptr(kk)=null;
                arg_array_rsm.r1_ptr(kk)=a_ptrs(1)->indexo
r(kk);
            end;
            else do;
                arg_array_rsm.q1_ptr(kk)=a_ptrs(1)->indexo
r(kk);
                arg_array_rsm.r1_ptr(kk)=null;
            end;
        end;



    /*Here we process calling name for most operations*/
setup5:   do kk=1 to limit;
            if op="R" then go to must_be_second;
            if arg_array_rsm.r1_ptr(kk)¬=null then do;
                if kk=1 then callname=addrel(a_ptrs(1)->p,
a_ptrs(1)->p->oset)->label_plate;
                else if addrel(a_ptrs(1)->indexor(kk),a_pt
rs(1)->indexor(kk)->oset)->label_plate¬=
                    callname then do;
                    /*Error handling$$$*/
                    free arg_array_rsm;
                    call GS_error$p(23,op,a_ptrs(1)->p,a_p
trs(1)->indexor(kk));
                end;
/*Error here is specification of inconsistant RSM names*/
            end;
            else do;
```

```
                    if ¬two_arg_sw|arg_array_rsm.r2_ptr(kk)=nu
ll|op="M"|op="O" then do;
                        if kk=1 then callname="rsm_wq";
                        else if callname¬="rsm_wq" then do;
                            free arg_array_rsm;
                            call GS_error$p(24,op,a_ptrs(1)->p
,a_ptrs(1)->indexor(kk));
                        end;
/*Error here is inconsistant RSM names*/
                    end;
                    else must_be_second:do;
                        if kk=1 then callname=addrel(a_ptrs(2)
->p,a_ptrs(2)->p->oset)->label_plate;
                        else if addrel(a_ptrs(2)->indexor(kk),
a_ptrs(2)->indexor(kk)->oset)->label_plate¬=
                            callname then do;
                            /*Error handling$$$*/
                            free arg_array_rsm;
                            call GS_error$p(25,op,a_ptrs(2)->p
,a_ptrs(2)->indexor(kk));
                        end;
/*Error here is inconsistant RSM names*/
                    end;
                end;
            end;


    /*Where it is proper, space is obtained for the new resu
lt relation*/
setup6:     if op="Q"|op="E"|op="M"|op="Z" then go to setup7;
            do kk=1 to limit;
                arg_array_rsm.return_arg(kk)=null;
                if arg_array_rsm.r1_ptr(kk)=null then go to no
_new_r;                                          & op¬="R"
                call hcs_$fs_search_get_wdir(addr(wdir),lwdir)
;
                call hcs_$make_seg(wdir,"","",access_code,arg_
array_rsm.return_arg(kk),err_code);
                if err_code>0 then do;/*Error handling$$$*/
                    free arg_array_rsm;
                    call GS_error$p(26,op,a_ptrs(1)->indexor(k
k),a_ptrs(2)->indexor(kk));
                end;
/*Error here is failure of make seg procedure*/

                if F_sw & op¬="R" then do;
                    F_sw=0b;
                    option_sw=1b;
                    go to no_name_needed;
                end;
                else do;
                    call ioa_$nnl("New relation created. Op is
 ¬1a. Relation 1 is ¬p. Give new name...",
```

```
                if S_xx then do;
                    if arg_array_rsm.a2_ptr(kk)^=null then
free a_ptrs(2)->indexor(kk)->quart;
                    else call bes$delentry_seg(a_ptrs(2)->
indexor(kk),err_code);
                end;
            end;
            if limit=1 then do;
                xfer=arg_array_rsm.return_arg(limit);
                free arg_array_rsm;
                a_ptrs(arg_no)->p=xfer;
                return;
            end;
            allocate return_structure;
            do mm=1 to limit;
                return_structure(mm)=arg_array_rsm.return_
arg(mm);
            end;
            free arg_array_rsm;
            a_ptrs(arg_no)->p=returnable_ptr;
            end;
```

oro.pl1 05/20/70 1025.3 edt Wed

```
%;
oro: procedure returns(pointer);

          %include mgr_declare;

nr:new_relation: entry returns(pointer);
          call cu_$arg_ptr(1,gen_purpose_ptr,lentty,err_code
);
          if err_code>0 then call GS_error$p(1,"N",null,null
);
/*Error here is null relation name*/
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          if arg_no¬=3 then call GS_error$p(2,"N",null,null)
;
/*Error here is incorrect argument specification for nr*/
          if a_ptrs(2+doset)->dt_code¬=13 then call GS_error
$p(3,"N",null,null);
/*Error here is non_pointer second argumetn for nr*/
          limit=1;
          allocate arg_array_rsm;
          op="N";
          if a_ptrs(2)->p=null then do;/*Error handling$$$*/

              free arg_array_dsm;
              call GS_error$p(4,op,null,null);
          end;
/*Error here is failure to specify an r for the rsm name*/
          callname=addrel(a_ptrs(2)->p,a_ptrs(2)->p->oset)->
label_plate;
          call hcs_$fs_search_get_wdir(addr(wdir),lwdir);
          call hcs_$make_seg(wdir,rel_name,rel_name,access_c
ode,arg_array_rsm.return_arg(limit),err_code);
          if err_code>0 then do;/*Error handling$$$*/
              free arg_array_rsm;
              call GS_error$c(5,op,rel_name,null);
          end;
/*Error here is failure in make segment procedure*/
          go to setup7;


ir:initiate_relation: entry returns(pointer);
          call cu_$arg_ptr(1,gen_purpose_ptr,lentty,err_code
);
          if err_code>0 then call GS_error$p(6,"Z",null,null
);
/*Error here is null relation name*/
          limit=1;
```

```
          limit-1,
          allocate arg_array_rsm;
          op="Z";
          call hcs_$fs_search_get_wdir(addr(wdir),lwdir);
          call hcs_$initiate(wdir,rel_name,rel_name,0,0,arg_
array_rsm.rl_ptr(limit),err_code);
          if err_code>0 & err_code¬=error_table_$segknown th
en do;/*Error handling$$$*/
              free arg_array_rsm;
              call GS_error$c(7,op,rel_name,null);
          end;
/*Error here is failure to initiate relation specified in ir
 operation*/
          callname=addrel(arg_array_rsm.rl_ptr(limit),arg_ar
ray_rsm.rl_ptr(limit)->oset)->label_plate;
          go to setup7;


sa:squash_array: entry returns(pointer);
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          if a_ptrs(1+doset)->dt_code¬=29 then call GS_error
$p(8,"A",null,null);
/*Error here is non array of pointers for first argument*/
          limit=1;
          allocate arg_array_rsm;
          limit=a_ptrs(1+doset)->top_bound;
          if a_ptrs(1+doset)->low_bound¬=1 then do;
              free arg_array_rsm;
              call GS_error$p(9,"A",a_ptrs(1)->p,null);
/*Error here is non one lowver bound*/
          end;
          alloc_length=0;
          alloc_order=a_ptrs(1)->p->order;
          do kk=1 to limit;
              alloc_length=alloc_length+a_ptrs(1)->indexor(k
k)->qlength;
              do jj=1 to alloc_order while (kk>1);
                  if a_ptrs(1)->indexor(kk)->cent(jj)¬=a_ptr
s(1)->p->cent(jj) then do;/*Error handling$$$*/
                      free arg_array_rsm;
                      call GS_error$p(10," ",a_ptrs(1)->p,a_
ptrs(1)->indexor(kk));
                  end;
/*Error here is inconsistant cent entries*/
              end;
          end;
          allocate quart set (gen_purpose_ptr);
          kik=1;
          arg_array_rsm.ql_ptr(kik)=gen_purpose_ptr;
          squash_sw=1b;
          new_place=0;
          do kk=1 to limit;
```

ero.pl1 05/20/70 1052.2 edt Wed

```
%;
ero: procedure returns(pointer);

          %include mgr_declare;


eq:expand_quart: entry returns (pointer);
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          if a_ptrs(1+doset)->dt_code¬=13 then call GS_error
$p(11," ",null,null);
/*Error here is non pointer argument*/
          limit=a_ptrs(1)->p->qlength;
          allocate return_structure;
          alloc_order=a_ptrs(1)->p->order;
          alloc_length=1;
          do kk=1 to limit;
              allocate quart set (return_structure(kk));
              do jj=1 to alloc_order;
                  return_structure(kk)->cent(jj)=a_ptrs(1)->
p->cent(jj);
                  return_structure(kk)->pent(jj)=a_ptrs(1)->
p->pent(jj);
                  return_structure(kk)->tent(1,jj)=a_ptrs(1)
->p->tent(kk,jj);
              end;
          end;
          free a_ptrs(1)->p->quart;
          a_ptrs(arg_no)->p=returnable_ptr;
          end;
```

ndo.pl1 05/20/70 1032.0 edt Wed

```
%;
ndo: procedure returns(pointer);

          %include mgr_declare;

gr:get_refno: entry returns(pointer);
          op="R";
          go to setup10;
sd:successor_data: entry returns(pointer);
          op="S";
          go to setup10;
gd:get_data: entry returns(pointer);
          op="D";
          go to setup10;


          /*Here check the general structure of arguments*/
setup10:  call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          if arg_no<2 then go to err_38;
          if a_ptrs(1+doset)->dt_code=13 then limit=1;
          else if a_ptrs(1+doset)->dt_code=29 then do;
              limit=a_ptrs(1+doset)->top_bound;
              if a_ptrs(1+doset)->low_bound~=1 then call GS_
error$p(37,op,null,null);
/*Error here is non zero lower bound on input array*/
          end;
          else  err_38:  call GS_error$p(38,op,null,null);
/*Error here is non pointer first argument*/
          if arg_no>6 then call GS_error$p(39,op,a_ptrs(1)->
p,null);
/*Error here is illegal number of arguments*/


          /*Here do the interpretation of the arguments*/
          if op="R" then do;
              if a_ptrs(2+doset)->dt_code=13 then ptr_sw=1b;

              if arg_no<4 then go to no_further_check;
              if a_ptrs(3+doset)->dt_code=13 then do;
                  if limit=1 then skip_3_sw=1b;
                  else call GS_error$p(40,op,a_ptrs(1)->p,a_
ptrs(3)->p);
/*Error here is inconsistant upper bounds on optional R argu
ment*/
              end;
```

```
            else if a_ptrs(3+doset)->dt_code=29 then do;
                if limit=a_ptrs(3+doset)->top_bound then s
kip_3_sw=1b;
                else call GS_error$p(40,op,a_ptrs(1)->p,a_
ptrs(3)->p);
/*Error here is inconsistant upper bounds on optional R argu
ment*/
            end;
no_further_check:if ¬ptr_sw then do;
                call cu_$arg_ptr(2,gen_purpose_ptr,as_len,
err_code);
                if err_code>0 then call GS_error$p(41,op,a
_ptrs(1)->p,null);
/*Error here is incorrect format for NAM field*/
                if as_len¬=4 then call GS_error$c(42,op,ba
sed_arg,a_ptrs(1)->p);
                nam=based_arg;
                nam_sw=1b;
            end;
        end;
        else do;
            call cu_$arg_ptr(2,gen_purpose_ptr,as_len,err_
code);
            if err_code=error_table_$noarg then go to no_m
ore;
            else if err_code>0 then call GS_error$p(43,op,
a_ptrs(1)->p,null);
/*Error here is format error in argument 2*/
            if as_len=4 then do;
                nam_sw=1b;
                nam=based_arg;
            end;
            else go to read_arg;
        end;


        iii=3;
next_one: if skip_3_sw then if iii=3 then iii=4;
        call cu_$arg_ptr(iii,gen_purpose_ptr,as_len,err_co
de);
        if err_code=error_table_$noarg then go to no_more;

        else if err_code>0 then call GS_error$p(43,op,a_pt
rs(1)->p,null);
/*Error here is format error in argument iii*/
read_arg: if as_len<4 then do;
            oplength=as_len;
            option_sw=1b;
            options=based_arg;
        end;
        else if as_len<33 then do;
            ent_sw=1b;
            ent=based_arg;
        end;
        else call GS_error$c(44,op,based_arg,a_ptrs(1)->p)
```

```
/*Error here is argument of string size greater than 32*/
          if iii=5 then go to no_more;
          else iii=iii+1;
          go to next_one;


          /*Here allocate the proper size argument array*/
no_more:  if option_sw & (index(options,"E")¬=0|index(option
s,"e")¬=0) then do;
              squash_sw=1b;
              if limit>1 then call GS_error$p(11,op,a_ptrs(1
)->p,null);
/*Error here is feeding array to conceptual expand_quart ope
ration*/
              else limit=a_ptrs(1)->p->qlength;
              allocate arg_array_dsm;
              limit=1;
          end;
          else allocate arg_array_dsm;


          /*Here set opcode and options slots*/
          arg_array_dsm.opcode=op;
          /* Here scan the options*/
          if option_sw then do;
              arg_array_dsm.d_or_i_option=" ";
              if (index(options,"D")¬=0|index(options,"d")¬=
0) then
                  arg_array_dsm.d_or_i_option="D";
              if op="R" & (index(options,"I")¬=0|index(optio
ns,"i")¬=0) then
                  arg_array_dsm.d_or_i_option="I";
              if squash_sw then arg_array_dsm.e_option="E";
              else arg_array_dsm.e_option=" ";
              do kik=1 to 3;
                  mm=index("123456789",substr(options,kik,1)
);
                  if mm¬=0 then do;
                      column_index=mm;
                      go to stop_col;
                  end;
              end;
stop_col:
          end;
          else arg_array_dsm.d_or_i_option,arg_array_dsm.e_o
ption=" ";


          /*Here set the return arg slot with given pointers
, if needed*/
          do kk=1 to limit;
              if skip_3_sw then arg_array_dsm.return_arg(kk)
=a_ptrs(3)->indexor(kk);
```

```
            else arg_array_dsm.return_arg(kk)=null;
        end;


        /*Here do consistancy checking on the pents or cen
ts*/
        if ¬nam_sw & ¬ptr_sw then do;
            iii=0;
            do kk=1 to limit;
                if a_ptrs(1)->indexor(kk)->quart.pent(colu
mn_index)¬=null then do;
                    if iii=0 then iii=kk;
                    else if a_ptrs(1)->indexor(kk)->quart.
pent(column_index)¬=
                        a_ptrs(1)->indexor(iii)->quart.pen
t(column_index) then do;
                        free arg_array_dsm;
                        call GS_error$p(45,op,a_ptrs(1)->i
ndexor(kk),a_ptrs(1)->indexor(iii));
/*Error here is inconsistant data type by non null pents*/
                    end;
                end;
                if substr(a_ptrs(1)->indexor(kk)->cent(col
umn_index),1,4)¬=
                    substr(a_ptrs(1)->p->cent(column_index
),1,4) then do;
                    free arg_array_dsm;
                    call GS_error$cc(46,op,a_ptrs(1)->p->c
ent(column_index),a_ptrs(1)->indexor(kk)->
                        cent(column_index));
                end;
/*Error here is inconsistant data types by cent name*/
            end;
        end;


        /*Here fill in first arguments*/
        do kk=1 to limit;
            arg_array_dsm.d_or_q_ptr(kk)=a_ptrs(1)->indexo
r(kk);
        end;


        /*Here construct the ddptr*/
        if ptr_sw then do;
            arg_array_dsm.data_data_ptr=a_ptrs(2)->p;
        end;
        else if ¬nam_sw then do;
            if iii=0 then do;
                iii=1;
                call hcs_$fs_search_get_wdir(addr(wdir),lw
dir);
                call hcs_$initiate(wdir,substr(a_ptrs(1)->
p->quart.cent(column_index),1,4),
```

```
                    substr(a_ptrs(1)->p->quart.cent(column_i
ndex),1,4),0,0,a_ptrs(1)->p->quart.pent(column_index),
                    err_code);
                if err_code=error_table_$segknown then do;
 end;
                else if err_code>0 then do;/*Error handlin
g$$$*/
                    free arg_array_dsm;
                    call GS_error$c(47,op,a_ptrs(1)->p->ce
nt(column_index),a_ptrs(1)->p);
                end;
/*Error here is failure in get seg ptr or initiate procedure
*/
            end;
            arg_array_dsm.data_data_ptr=a_ptrs(1)->indexor
(iii)->quart.pent(column_index);
        end;
        else do;
            call hcs_$fs_search_get_wdir(addr(wdir),lwdir)
;
            call hcs_$initiate(wdir,substr(nam,1,4),substr
(nam,1,4),
                0,0,arg_array_dsm.data_data_ptr,err_code);

            if err_code=error_table_$segknown then do; end
;
            else if err_code>0 then do;/*Error handling$$$
*/
                free arg_array_dsm;
                call GS_error$c(48,op,nam,a_ptrs(1)->p);
            end;
/*Error here is failure in get seg ptr or intitiate procedur
e*/
        end,


        /*Here set callname and issue call to dsm*/
        callname=addrel(data_data_ptr,data_data_ptr->oset)
->label_plate;
        call hcs_$make_ptr("",callname,callname,xfer,err_c
ode);
        if err_code>0 then do;
            free arg_array_dsm;
            call GS_error$c(49,op,callname,null);
        end;
        call cu_$ptr_call(xfer,arg_ptr);


        /*Here do post call processing like filling in ret
urn args and ent*/
        do kk=1 to limit;
            if (op="S"|op="R") & ent_sw then do;
                arg_array_dsm.return_arg(kk)->quart.cent(c
olumn_index)=ent;
```

```
[illegible]
end;
if [illegible] then do;
    [illegible]_array_[illegible].return_arr(limit);
    [illegible] [illegible]_array_dsm;
    [illegible](arr_no)->p=xfer;
    [illegible];
end;
allo [illegible] return_structure;
do [illegible] limit;
    [illegible]_structure(mm)=a[illegible]_array_dsm(mm).
return_[illegible];
end;
free [illegible]_array_dsm;
[illegible](arr_no)->p=returnable_ptr;
end;
```

odo.pl1 05/20/70 1044.3 edt Wed

```
%;
odo: procedure returns (pointer);

          %include mgr_declare;


nd:new_data_type: entry returns(pointer);
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          call cu_$arg_ptr(2,gen_purpose_ptr,lentty,err_code
);
          if err_code>0 then call GS_error$p(28,"N",null,nul
l);
          else callname=rel_name;
/*Error here is non label second arg*/
          limit=1;
          allocate arg_array_dsm;
          do;
              call cu_$arg_ptr(1,gen_purpose_ptr,dlen,err_co
de);
              if err_code>0 then do;/*Error handling$$$*/
                  free arg_array_dsm;
                  call GS_error$p(29,"N",null,null);
              end;
/*Error here is incorrect first argument*/
              if dlen¬=4 then do;/*Error handling$$$*/
                  free arg_array_dsm;
                  call GS_error$c(30,"N",based_nam,null);
              end;
/*Error here is incorrect length for data type name*/
              call hcs_$fs_search_get_wdir(addr(wdir),lwdir)
;
              call hcs_$make_seg(wdir,based_nam,based_nam,ac
cess_code,arg_array_dsm.data_data_ptr,err_code);
              if err_code>0 then do;/*Error handling$$$*/
                  free arg_array_dsm;
                  call GS_error$c(31,"N",based_nam,null);
              end;
/*Error here is failure in make seg procedure*/
              arg_array_dsm.opcode="N";
          end;
          arg_array_dsm.high_bound=limit;
          call hcs_$make_ptr("",callname,callname,xfer,err_c
ode);
          if err_code>0 then do;
              free arg_array_dsm;
              call GS_error$c(49,arg_array_dsm.opcode,callna
```

arg_array_dsm.d_or_i_option,
arg_array_dsm.e_option=" ";

```
me,null);
/*Error here is illegal callname*/
          end;
          call cu_$ptr_call(xfer,arg_ptr);
          a_ptrs(arg_no)->p=data_data_ptr;
          return;

kd:kill_data_type: entry returns(pointer);
          limit=1;
          call cu_$arg_list_ptr(param_ptr);
          if display_if_8=8 then doset=arg_no+1;
          else doset=arg_no;
          if a_ptrs(1+doset)->dt_code=520|a_ptrs(1+doset)->d
t_code=522 then do;
               call cu_$arg_ptr(1,gen_purpose_ptr,lentty,err_
code);
               if err_code=error_table_$noarg then call GS_er
ror$p(32,"K",null,null);
               else if err_code=0 then do;
                    if lentty¬=4 then call GS_error$c(33,"K",r
el_name,null);
                    call hcs_$fs_search_get_wdir(addr(wdir),lw
dir);
                    call hcs_$delentry_file(wdir,based_nam,err
_code);
                    if err_code>0 then call GS_error$c(34,"K",
based_nam,null);
/*Error here is failure in initiate or in mget seg ptr*/
               end;
          end;
          else if a_ptrs(1+doset)->dt_code=13 then do;
               if a_ptrs(1)->p->quart.pent(limit)=null then d
o;
                    call hcs_$fs_search_get_wdir(addr(wdir),lw
dir);
                    call hcs_$initiate(wdir,substr(a_ptrs(1)->
p->quart.cent(limit),1,4),
                         substr(a_ptrs(1)->p->quart.cent(limit)
,1,4),0,0,gen_purpose_ptr,err_code);
                    if err_code=error_table_$segknown then do;
 end;
                    else if err_code>0 then call GS_error$c(35
,"K",a_ptrs(1)->p->cent(limit),a_ptrs(1)->p);
/*Error here is failure in initiate or in get seg ptr*/
               end;
               else gen_purpose_ptr=a_ptrs(1)->p->quart.pent(
limit);
               call hcs_$delentry_seg(gen_purpose_ptr,err_cod
e);
               if err_code>0 then call GS_error$p(36,"K",gen_
purpose_ptr,null);
/*Error here is failure in delete seg procedure*/
          end;
```

<s>GS_error.pl1 05/20/70 1059.4 edt Wed

```
GS_error: procedure;
          declare (p,pr) entry (fixed bin(17),char(1),pointe
r,pointer);
          declare (c,cr) entry (fixed bin(17),char(1),char(*
),pointer);
          declare (cc,ccr) entry (fixed bin(17),char(1),char
(*),char(*));
          declare code fixed bin(17);
          declare op char(1);
          declare (p1,p2) pointer;
          declare (c1,c2) char(*);
          declare err_code fixed bin(17);
          declare puse pointer;
          declare pi_label label static;
          declare GS_error$pi entry external;
          declare buffer char(120) initial(" ");
          declare num_items fixed bin(35) initial(120);
          declare ascode char(12);
          declare (cond_flag,p_sw,r_sw,c_sw,cc_sw) fixed bin
(1) initial(0b);
          declare 1 quart based (quart_ptr),
              2 ID fixed bin(35),
              2 qlength fixed bin(35),
              2 order fixed bin(35),
              2 cent(1 refer (order)) char(32),
              2 pent(1 refer (order)) pointer,
              2 tent(1 refer (qlength),1 refer (order)) fixe
d bin(35);
pr:       entry (code,op,p1,p2);
          r_sw=1b;
p:        entry (code,op,p1,p2);
          p_sw=1b;
          go to message;
cr:       entry (code,op,c1,p1);
          r_sw=1b;
c:        entry(code,op,c1,p1);
          c_sw=1b;
          go to message;
ccr:      entry (code,op,c1,c2);
          r_sw=1b;
cc:       entry(code,op,c1,c2);
          cc_sw=1b;
message: call ioa_("GS: Error ¬5d  Internal opcode ¬a",code,
op);
          if cc_sw then call ioa_("    Name 1: ¬a  Name 2: ¬
a",c1,c2);
          else if c_sw then do;
              call ioa_$nnl("    Name 1: ¬a ",c1);
              if p1¬=null then call ioa_(" Pointer 1: ¬p",p1
```

```
);
            else call ioa_(" ");
        end;
        if p_sw then do;
            if p1¬=null then call ioa_$nnl("    Pointer 1:
 ¬p ",p1);
            if p2¬=null then call ioa_$nnl("    Pointer 2:
 ¬p ",p2);
            if p1¬=null|p2¬=null then call ioa_(" ");
        end;
        call cv_bin_(code,ascode,10);
        call ioa_$nnl("    ");
        call print("GS_err_messages",ascode,ascode);
new_try:   if r_sw then call ioa_$nnl("    Type procedure na
me, return, or quit...");
        else call ioa_$nnl("    Type procedure name, or qu
it...");
        call ios_$read_ptr(addr(buffer),num_items,kk);
        if r_sw & substr(buffer,1,kk-1)="return" then retu
rn;
        if substr(buffer,1,kk-1)="quit" then call signal_(
"GOLD_STAR");
        call hcs_$make_ptr("",substr(buffer,1,kk-1),substr
(buffer,1,kk-1),puse,err_code);
        if err_code>0 then call ioa_("    No such procedur
e can be found!");
        else call cu_$ptr_call(puse);
        go to new_try;
        end;
```

1 Argument for nr operation not found.
2 nr operation does not have two arguments plus return argument(non_functional invocation is illegal).
3 Second argument of nr operation is not a pointer.
4 Second argument of nr operation is a null pointer.
5 Make segment procedure in nr operation failed - name provided is attempted relation name.
6 Argument not found for ir operation.
7 Initiate procedure failed for ir operation - name provided is attempted name.
8 Argument for sa operation is not an array of pointers.
9 Low bound of argument(pointer 1) is not one.
10 CENT inconsistancy found between elements of array of pointers in sa operation(pointer 1 and 2).
11 Argument in eq operation must be pointer.
12 Too few arguments-must be at least one plus return for functional invocation.
13 First argument must be pointer or array of pointers.
14 Low bound of first argument(pointer 1) must be one.
15 Too many arguments(pointer 1 is first one).
16 All arguments must be pointers if first one is(pointer 1).
17 All arguments must be arrays of pointers if first one is(pointer 1).
18 Upper bound of input argument arrays(pointers 1 and 2) are not the same.
19 Operation cannot have more than one argument(pointer 1).
20 Operation must have more than one argument(pointer 1).
21 Successor cannot have relation as second argument(arg 1=pointer 1, arg 2=pointer 2).
22 First operand cannot be null.
23 First argument is array of relation pointers calling for inconsistant RSM(pointers 1 and 2).
24 Second argument is an illegal array of quart and relation pointers(pointers 1 and 2).
25 Second argument is array of relation pointers calling for inconsistant RSM(pointers 1 and 2).
26 Make segment procedure failed - arguments one and two are pointers 1 and 2.
27 Change name procedure has failed on segment of pointer 1.
28 Illegal second argument for nd operation.
29 Illegal first argument for nd operation.
30 Data type name (Name 1) in nd operation is not four characters.
31 Make segment procedure has failed during nd operation(name attempted was Name 1).
32 No argument given for kd operation.
33 Type(Name 1) indicated in kd operation not four characters.
34 Delete segment procedure failed during kd operation(name att

```
   78
   79
   80
   81
   82
   83
   84
   85
   86
   87
   88
   89
   90
   91
   92
   93
   94
   95
   96
   97
   98
   99
100Overlapping CENT in cp operation between Pointer 1 and 2. Re
   turn yeilds null pointer.
101Orders of quarts (pointers 1 and 2) are not equal. Return ye
   ilds null result.
102CENT of quarts (pointers 1 and 2) do not overlap. Return yei
   lds null result.
103Width of equivalence vector(Pointer 2) must be equal to orde
   r of quart(Pointer 1).
   104
   105
```

```
gsdb:   procedure;
        declare 1 dsm based (inptr),
            2 dtop fixed bin(17),
            2 cindex fixed bin(17),
            2 dop char(1),
            2 di char(1),
            2 eopt char(1),
            2 ddptr pointer,
            2 dqptr (1 refer (dtop)) pointer,
            2 dretptr (1 refer (dtop)) pointer;
        declare 1 rsm based (inptr),
            2 rtop fixed bin(35),
            2 rop char(1),
            2 q1 (1 refer (rtop)) pointer,
            2 r1 (1 refer (rtop)) pointer,
            2 q2 (1 refer (rtop)) pointer,
            2 r2 (1 refer (rtop)) pointer,
            2 ep (1 refer (rtop)) pointer,
            2 rretptr (1 refer (rtop)) pointer;
        declare 1 quart based (inptr),
            2 ID fixed bin(35),
            2 qlen fixed bin(35),
            2 qorder fixed bin(35),
            2 cent (1 refer (qorder)) char(32),
            2 pent (1 refer (qorder)) pointer,
            2 tent (1 refer (qlen),1 refer (qorder)  fixed
bin(35);
        declare 1 entvector based (inptr),
            2 width fixed bin(35),
            2 ents (1 refer (width)) char(32);
        declare gsdb$pi entry external;
        declare (d,r,q,e) entry (pointer) external;
        declare call_sw fixed bin(1) initial(0b);
        declare pi_label label static;
        declare op char(4) initial("    ");
        declare buffer char(132);
        declare (baseptr,addrel,min,substr,index) builtin;

        declare cv_oct_ entry external returns (fixed bin(
35));
        declare woffset fixed bin(35) initial(0);
        declare wptr pointer initial (null);
        declare (inptr,aptr) pointer;
        declare (mm,kk,ii,orplace,endplace) fixed bin(17);


        call ioa_("ASK!");
        call_sw=1b;
        pi_label=read;
```

```
read:       call ios_$read_ptr(addr(buffer),132,kk);
            op=substr(buffer,1,1);
            if op="." then return;
            if op¬="d" & op¬="r" & op¬="q" & op¬="e" then do;
                call ioa_("?");
                go to read;
            end;
            do mm=2 to kk-1;
                if substr(buffer,mm,1)¬=" " then go to start_n
o;
            end;
            go to no_offset;
start_no:   orplace=index(buffer,"|");
            if orplace=0 then do;
                orplace=mm-1;
                go to offset_only;
            end;
            wptr=baseptr(cv_oct_(substr(buffer,mm,orplace-mm))
);
offset_only: do ii=orplace+1 to kk-1;
                if substr(buffer,ii,1)=" " then do;
                    endplace=ii;
                    go to end_found;
                end;
            end;
            endplace=kk-1;
end_found:  mm=endplace+1-orplace;
            if mm=0 then go to no_offset;
            woffset=cv_oct_(substr(buffer,orplace+1,mm));
no_offset:  inptr=addrel(wptr,woffset);
            go to output;


d:          entry(aptr);
            op="d";
            go to labset;
r:          entry(aptr);
            op="r";
            go to labset;
q:          entry(aptr);
            op="q";
            go to labset;
e:          entry(aptr);
            op="e";
labset:     pi_label=back;
            inptr=aptr;
output:     if inptr=null then do;
                call ioa_("NULL!");
                go to done;
            end;
            call condition_("program_interrupt",gsdb$pi);
            call ioa_("AT ¬p",inptr);
            if op="q" then do;
```

```
            do ii=1 to qorder;
                call ioa_("¬16a  ¬p",cent(ii),pent(ii));
            end;
            do ii=1 to qlen;
                do mm=1 to qorder;
                    call ioa_$nnl("¬8d ",tent(ii,mm));
                end;
                call ioa_(" ");
            end;
        end;
        else if op="r" then do;
            call ioa_("Bound ¬3d  Op  ¬1a",rtop,rop);
            call ioa_("Q1       R1       Q2       R2
 EP       RET");
            do ii=1 to rtop;
                call ioa_("¬p ¬p ¬p ¬p ¬p ¬p",q1(ii),r1(ii
),q2(ii),r2(ii),ep(ii),rretptr(ii));
            end;
        end;
        else if op="d" then do;
            call ioa_("Bound ¬3d  Op  ¬1a  DI option  ¬1a
 E option  ¬1a  C option  ¬1d  DDptr ¬p",
                dtop,dop,di,eopt,cindex,ddptr);
            call ioa_("D or Q   RETURN");
            do ii=1 to dtop;
                call ioa_("¬p ¬p",dqptr(ii),dretptr(ii));
            end;
        end;
        else do ii=1 to width;
            call ioa_("¬16a",ents(ii));
        end;
done:   if call_sw then go to read;
        else
back:        return;


pi:     entry;
        go to pi_label;
        end;
```

dsm_astring.pl1 06/01/70 2108.5 edt Mon

```
dsm_astring: procedure (dsm_args_ptr);

  dcl data_seg area (area_size) based (seg_ptr);

  dcl 1 quart based (quart_ptr),
        2 id fixed binary (35),
        2 length fixed bin (35),
        2 order fixed bin (35),
        2 cent char (32),
        2 pent ptr,
        2 tent fixed bin (35);

  dcl 1 quart_proper based (uart_ptr),
           2 qid fixed bin (35),
           2 qlength fixed bin (35),
           2 qorder fixed bin (35),
           2 qcent (qorder) char (32),
           2 qpent (qorder) ptr,
           2 qtent (qlength,qorder) fixed bin (35);
  dcl 1 mt_quart static,
        2 mtid fixed bin (35) initial (0),
        2 mtlength fixed bin (35) initial (0),
        2 mtorder fixed bin (35) initial (1),
        2 mtcent char (32),
        2 mtpent ptr;

  dcl 1 dsm_args based (dsm_args_ptr),
        2 upper_bound fixed binary (17),
        2 ci fixed bin (17),
        2 op char (1),
        2 di char (1),
        2 e char (1),
        2 ddptr ptr,
        2 d_or_q_ptr (upper_bound) ptr,
        2 return_ptr (upper_bound) ptr;

  dcl 1 type based (type_ptr),
        2 dsm_name char (16),
        2 data_type char (32),
        2 num_free_cells fixed bin (35),
        2 num_entries fixed bin (35),
        2 max_length fixed bin (35),
        2 first_entry offset (data_seg),
        2 successor_chain_head offset (data_seg);
```

```
   dcl 1 item based (iptr),
         2 lptr offset (data_seg),
         2 rptr offset (data_seg),
         2 succ offset (data_seg),
         2 flag fixed binary (35),
         2 refno fixed bin (35),
         2 string_structure,
           3 string_length fixed binary (35),
           3 string character (max_length);

   dcl 1 area_seg based (seg_ptr),
         2 first_off  offset (data_seg),
         2 curr_len   offset (data_seg),
         2 next_off offset (data_seg);

   dcl 1 pseudo_string_array based (str_ptr),
         2 pseudo_length fixed binary (35),
         2 input_or_return_string  char (mlen_int);

   dcl (ptr,new_ptr,succ_ptr,type_ptr,new_seg_ptr,
        quart_ptr,seg_ptr,iptr,str_ptr) ptr static;

   dcl dsm_args_ptr ptr;

   dcl (area_size,i,ii,found,mlen_int,diff,differ,turn,presen
ted_loop_refno,stored_loop_refno,
        iii,ik,k,grech) fixed binary (35) static;

   dcl (max_depth_of_tree initial (34),left initial (10),righ
t initial (20)  fixed binary (35) static;

   dcl return_label label (insert_return,datum_return,refno_r
eturn,successor_return,
                           delete_by_refno_return,delete_by_d
atum_return,r_,d_,s_,i_,rd,dd);

   dcl dbd_label (-1:1) label (begin_dsm_astring) static;

   dcl dbr_label (-1:1) label (begin_dsm_astring) static;

   dcl succ_label (-1:1) label (begin_dsm_astring) static;

   dcl insert_label (1:1) label (begin_dsm_astring) static;

   dcl ins_comp_label (-1:0) label (begin_dsm_astring) static
;

   dcl ename char (32), dirname char (168) static;

   dcl (presented_loop_string,stored_loop_string) character (
168) varying static;

   dcl (exp,dori,opcode) char (1) varying static;
```

```
   dcl tree_restructure entry (pointer,pointer,pointer),
       (area_$redef,area_) entry  (fixed binary (17),pointer)
;

   dcl (null,nullo,substr,divide,addrel,min,addr) builtin;

   dcl (off,curr,next,code) fixed binary (35) static;

   dcl (glich,last_left,last_right) offset (data_seg) static;


   dcl (dbd_lab_sw,dbr_lab_sw,ins_lab_sw,succ_lab_sw) fixed b
in (35) static initial (0);


/*****************************************************************
***********************************************/


begin_dsm_astring:

  exp=dsm_args.e;
  dori=dsm_args.di;
  opcode=dsm_args.op;
  seg_ptr=dsm_args.ddptr;
  type_ptr=first_off;

    if opcode="D" then   /* input is refnos */
      if dori=" " then go to get_datum;
        else if dori="D" then go to delete_by_refno;
          else go to op_error;

    else if opcode="D" then     /* data strings are input */

      if dori=" " then go to get_refno;
      else if dori ="D" then go to delete_by_datum;
      else if dori="I" then go to insert;
        else go to op_error;

      else if opcode="S" then
        if dori =" " then go to successor;
        else go to op_error;


    else if opcode="""" then
      if (dori=" " & exp=" ") then go to new_data_type;
      else go to op_error;


    else go to op_error;


/************************************/
```

```
new_data_type:
          do ii = 1 to upper_bound;
            call area_(1024,seg_ptr);
            call hcs_$fs_get_path_name(seg_ptr,dirname,grech
,ename,code);
            allocate type set (type_ptr) in (data_seg);
            num_entries=0;
            num_free_cells=0;
            call ioa_("what is max length for character stri
ngs for ^a>^a ?^/",substr(dirname,1,grech),ename);
            call read_list_(grech);
            max_length=grech;
            successor_chain_head=nullo;
            data_type=ename;
            dsm_name="dsm_astring";
            first_entry=nullo;
            end;
         return;


/*************************************/


get_datum:
        do ii = 1 to upper_bound;
          return_label=datum_return;
          go to refno_prologue;
         datum_return:
          if found=1 & ptr->flag=1 then do;
            return_label=d_;
            go to string_alloc;
           d_:
            pseudo_length=ptr->string_length;
            input_or_return_string=substr(ptr->string,1,ptr-
>string_length);
            end;
          else do;
             dsm_args.return_ptr(ii)=addr(mt_quart);
           mtcent=data_type;
           mtpent=seg_ptr;
           end;
      end;
    return;


/*************************************/
```

```
get_refno:
        do ii = 1 to upper_bound;
          return_label=refno_return;
          go to datum_prologue;
         refno_return:
          if found=1 & ptr->flag=1 then do;
            return_label=r_;
            go to quart_alloc;
           r_:
            tent=ptr->refno;
            end;
          else do;
            dsm_args.return_ptr(ii)=addr(mt_quart);
            mtcent=data_type;
            mtpent=seg_ptr;
            end;
      end;
    return;


/*************************************/


successor:
        if succ_lab_sw=0 then do;
          succ_lab_sw=1;
          succ_label (-1)=succ_lt_0;
          succ_label (0) =succ_eq_0;
          succ_label (1)=succ_gt_0;
          end;
        do ii = 1 to upper_bound;
          return_label=s_;
          go to quart_alloc;
         s_:
          succ_ptr=quart_ptr;
          return_label=successor_return;
          go to refno_prologue;
         successor_return:
          go to succ_label(found);


         succ_eq_0:
          if turn=left then do;
            if last_right=null() then do;
              ptr=successor_chain_head;
              if ptr->flag=1 then do;
                succ_ptr->tent=ptr->refno;
                go to successor_end;
                end;
              else go to succ_gt_0;
              end;
            else ptr=last_right;
            end;
```

```
          succ_gt_0:
           if ptr->succ=null() then
            succ_lt_0: do;
              return_ptr(ii)=addr(nt_quart);
              ntcent=data_type;
              ntscent=seg_ptr;
              go to successor_end;
              end;
           else if ptr->succ->flag=1 then do;
              succ_ptr->tent=ptr->succ->refno;
              go to successor_end;
              end;
           else do;
              plic=ptr->succ;
              ptr=plic;
              go to succ_gt_0;
              end;

     successor_end: end;
         return;


/*************************************/


insert:
     if ins_lab_siz=0 then do;
       ins_lab_siz=1;
       insert_label(-1)=ins_lt_0;
       insert_label(0)=ins_eq_0;
       insert_label(1)=ins_gt_0;
       ins_comp_label(-1)=ins_comp_lt_0;
       ins_comp_label(0)=ins_comp_eq_0;
       end;
     do ii = 1 to upper_bound;
       return_label=insert_return;
       go to data_prologue;
      insert_return:
       return_label=i_;
       go to quart_alloc;
      i_:
       go to insert_label(found);


      ins_lt_0: ins_eq_0:
       allocate item set (intr) in (data_seg);
       if intr=null then do;
         call area_$redef (unspec(substr(bit(unspec(next_off)
,18),1,18))+1025,seg_ptr);
         go to ins_lt_0;
         end;
```

```
        lptr=nullo;
        rptr=nullo;
        flag=1;
        string_length=pseudo_length;
        string=substr(input_or_return_string,1,min(max_length,
pseudo_length));
        go to ins_comp_label (found);

      ins_comp_lt_0:
        succ=nullo;
        refno=17179869184;
        tent=17179869184;
        first_entry=iptr;
        successor_chain_head=iptr;
        go to ins_incr;

      ins_comp_eq_0:
        diff=max_depth_of_tree-iii;
        differ=1;
        do ik=1 to diff;
          differ=2*differ;
          end;
        if turn=right then do;
          ptr->rptr=iptr;
          iptr->succ=ptr->succ;
          ptr->succ=iptr;
          iptr->refno=ptr->refno+differ;
          tent=iptr->refno;
          go to ins_incr;
          end;
        else do;
          iptr->succ=ptr;
          ptr->lptr=iptr;
          iptr->refno=ptr->refno-differ;
          tent=iptr->refno;
          if last_right=nullo then successor_chain_head=iptr;
          else last_right->succ=iptr;
          go to ins_incr;
          end;

      ins_gt_0:
        tent=ptr->refno;
        if ptr->flag=0 then do;
          ptr->flag=1;
          num_free_cells=num_free_cells-1;
          end;
        else go to insert_end;
      ins_incr:
        num_entries=num_entries+1;

  insert_end: end;
    return;
```

```
/************************************/


delete_by_refno:
       if dbr_lab_sw=0 then do;
         dbr_label(-1)=dbr_lt_0;
         dbr_label(0)=dbr_eq_0;
         dbr_label(1)=dbr_gt_0;
         dbr_lab_sw=1;
         end;
       do ii = 1 to upper_bound;
         return_label=delete_by_refno_return;
         go to refno_prologue;
        delete_by_refno_return:
         go to dbr_label (found);

        dbr_lt_0: dbr_eq_0:
         dsm_args.return_ptr(ii)=addr(mt_quart);
         mtcent=data_type;
         mtpent=seg_ptr;
         go to delete_by_refno_end;


        dbr_gt_0:
         if ptr->flag=0 then go to dbr_eq_0;
         else;
         ptr->flag=0;
         num_entries=num_entries-1;
         num_free_cells=num_free_cells+1;
         return_label=rd;
         go to string_alloc;
        rd:
         pseudo_length=ptr->string_length;
         input_or_return_string=substr(ptr->string,1,pseudo_l
ength);


       delete_by_refno_end: end;
       if 2*num_free_cells > num_entries then go to tree_rebu
ild;
       return;


/************************************/
```

```
delete_by_datum:
       if dbd_lab_sw=0 then do;
         dbd_lab_sw=1;
         dbd_label(-1)=dbd_lt_0;
         dbd_label(0)=dbd_eq_0;
         dbd_label(1)=dbd_gt_0;
         end;
       do ii = 1 to upper_bound;
         return_label=delete_by_datum_return;
         go to datum_prologue;
        delete_by_datum_return:
         go to dbd_label (found);

        dbd_lt_0: dbd_eq_0:
         dsm_args.return_ptr(ii)=addr(mt_quart);
         mtcent=data_type;
         mtpent=seg_ptr;
         go to delete_by_datum_end;


        dbd_gt_0:
         if ptr->flag=0 then go to dbd_eq_0;
         else;
         ptr->flag=0;
         num_entries=num_entries-1;
         num_free_cells=num_free_cells+1;
         return_label=dd;
         go to quart_alloc;
        dd:
         tent=ptr->refno;


       delete_by_datum_end: end;
       if 2*num_free_cells > num_entries then go to tree_rebu
ild;

    return;


/*************************************/
```

```
    refno_prologue:
      if exp="F" then quart_ptr=d_or_q_ptr(1);
      else quart_ptr=d_or_q_ptr(ii);
      plen_int=max_length;


    search_by_refno:
      turn,found=0;
      if first_entry=nullo then do;
          found=-1;
          go to return_label;
          end;
      else ptr=first_entry;
          if exp="F" then presented_loop_refno=quart_proper.
qtent(ii,ci);
          else presented_loop_refno=quart.tent;
          last_left,last_right=nullo;
           do iii=1 to max_depth_of_tree;
             stored_loop_refno=ptr->item.refno;
             if presented_loop_refno < stored_loop_refno then
 do;
               if ptr->lptr^=nullo then do;
                last_left=ptr;
                glich=ptr->lptr;
                ptr=glich;
                 go to search_by_refno_end;
                 end;
               else do;
                 turn=left;
                 go to return_label;
                 end;
              end;
             if presented_loop_refno > stored_loop_refno then
 do;
               if ptr->rptr^=nullo then do;
                last_right=ptr;
                 glich=ptr->rptr;
                ptr=glich;
                 go to search_by_refno_end;
                 end;
               else do;
                 turn=right;
                 go to return_label;
                 end;
               end;
             if presented_loop_refno = stored_loop_refno then
 do;
               found=1;
               go to return_label;
               end;
           search_by_refno_end: end;
                go to tree_rebuild;

/************************************/
```

```
    datum_prologue:
      str_ptr=dsm_args.d_or_q_ptr(ii);
      mlen_int=max_length;

    search_by_datum:
      found,turn=0;
      if first_entry=nullo then do;
          found=-1;
          go to return_label;
          end;
      else ptr=first_entry;
          presented_loop_string=substr(input_or_return_strin
g,1,pseudo_length);
          last_left,last_right=nullo;
          do iii=1 to max_depth_of_tree;
            stored_loop_string=substr(ptr->item.string,1,ptr
->string_length);
            if presented_loop_string < stored_loop_string th
en
             less_than: do;
              if ptr->lptr=nullo then do;
                last_left=ptr;
                glich=ptr->lptr;
                ptr=glich;
                go to search_by_datum_end;
                end;
              else do;
                turn=left;
                go to return_label;
                end;
              end;
            else if presented_loop_string > stored_loop_stri
ng then
            greater_than: do;
              if ptr->rptr=nullo then do;
                last_right=ptr;
                glich=ptr->rptr;
                ptr=glich;
                go to search_by_datum_end;
                end;
              else do;
                turn=right;
                go to return_label;
                end;
               end;
            else if presented_loop_string = stored_loop_stri
ng then do;
                found=1;
                go to return_label;
                end;
        search_by_datum_end: end;
         go to tree_rebuild;
```

```
/*********************************************/

    quart_alloc:
      allocate quart_set (quart_ptr);
      dsn_args.return_ptr(ii)=quart_ptr;
      id=0;
      cert=data_type;
      pcnt=seg_ptr;
      length=1;
      order=1;
      go to return_label;


    string_alloc:
      nlen_int=s_length;
      allocate based_string_array set (str_ptr);
      dsn_args.return_ptr(ii)=str_ptr;
      go to return_label;

/*********************************************/



    on_error:  [illegible] to error_node;
                 dsn_args.return_ptr(ii)=addr(nt_quart);
                 ntcert=data_type;
                 ntpcnt=seg_ptr;
                 end;
              return;


    tree_rebuild: call tree_restructure (seg_ptr,new_seg_ptr
,quart_ptr);
                    return;


end dsn_astring;
```

dsm_integer.pl1 06/01/70 1811.1 edt Mon

```
dsm_integer: proc (dsm_args_ptr);

dcl data_seg area (1024) based (seg_ptr);

dcl first_off offset (data_seg) based (seg_ptr);

dcl 1 dsm_args based (dsm_args_ptr),
            2 upper_bound fixed bin (17),
            2 ci fixed bin (17),
            2 op char (1),
            2 di char (1),
            2 e char (1),
            2 ddptr ptr,
            2 d_or_q_ptr (upper_bound) ptr,
            2 return_ptr (upper_bound) ptr;

dcl 1  type based (type_ptr),
            2 dsm_name char (32),
            2 data_type char (16);

dcl 1 quart based (quart_ptr),
            2 id fixed bin (35) initial (0),
            2 length fixed bin (35) initial (1),
            2 order fixed bin (35) initial (1),
            2 cent char (32),
            2 pent ptr,
            2 tent fixed bin (35);

  dcl 1 quart_proper based (quart_ptr),
            2 qid fixed bin (35),
            2 qlength fixed bin (35),
            2 qorder fixed bin (35),
            2 qcent (qorder) char (32),
            2 qpent (qorder) ptr,
            2 qtent (qlength,qorder) fixed bin (35);
dcl 1 mt_quart static,
            2 id fixed bin (35) initial (0),
            2 mtlength fixed bin (35) initial (0),
            2 mtorder fixed bin (35) initial (1),
            2 mtcent char (32),
            2 mtpent ptr;

dcl 1 pseudo_string_array based (str_ptr),
            2 pseudo_length fixed bin (35),
            2 input_or_return_string char (mlen_int);
```

```
dcl (jj,ll,mlen_int,temp,sign,sloc,i,ii) fixed bin (35);

dcl c11 (12) char (1) unaligned,
    c112 char (12) defined c11(1) unaligned;

dcl (quart_ptr,str_ptr,succ_ptr,dsm_args_ptr,con_ptr,seg_ptr
,type_ptr) pointer;
declare exp char(1) varying;
dcl opcode char (1) varying;

dcl dir char (168), ename char (32), grech fixed bin (35), c
ode fixed bin (17);

dcl (divide,substr,null,addr) builtin;

dcl rem fixed bin (35) based (con_ptr),
    false char (4) based (con_ptr);

dcl pot (0:11) fixed bin (35) initial (1,10,100,1000,10000,1
00000,1000000,10000000,
                                       100000000,1000000000,
10000000000,100000000000) static;

/*************************************************************
**********/

opcode=dsm_args.op;
seg_ptr=ddptr;
type_ptr=first_off;

if opcode="R" then go to get_refno;
else if opcode="D" then go to get_datum;
else if opcode="S" then go to successor;
else if opcode="N" then go to new_data_type;
else ii=0;
loop_2:
            ii=i+1;
            dsm_args.return_ptr(ii)=addr(mt_quart);
            seg_ptr=ddptr;
            type_ptr=first_off;
            mtpent=seg_ptr;
            mtcent=data_type;
            if ii<upper_bound then go to loop_2;
    return;


/*************************************************************
**********/
```

```
successor:
            ii=0;
       loop_3:
            ii=ii+1;
            allocate quart set (succ_ptr);
            succ_ptr->cent=data_type;
            succ_ptr->pent=seg_ptr;
            return_ptr(ii)=succ_ptr;
            if exp="E" then quart_ptr=d_or_q_ptr(1);
            else quart_ptr=d_or_q_ptr(ii);
            succ_ptr->cent=quart_ptr->cent;
            succ_ptr->pent=quart_ptr->pent;
            if exp="E" then succ_ptr->tent=1+quart_ptr->tent(i
ii,ci);
            else succ_ptr->tent=1+quart_ptr->tent;
            if ii<upper_bound then go to loop_3;
            return;


/*************************************************************
**********/


get_datum:
            on fixedoverflow begin;
                        dsm_args.return_ptr(ii)=addr(mt_quart);
                        mtpent=seg_ptr;
                        mtcent=data_type;
                        end;

            ii=0;
       loop_7:
            ii=ii+1;
            if exp="E" then quart_ptr=d_or_q_ptr(1);
            else quart_ptr=d_or_q_ptr(ii);
            mlen_int=12;
            allocate false set (con_ptr);
            allocate pseudo_string_array set (str_ptr);
            return_ptr(ii)=str_ptr;
            if exp="E" then temp=qtent(ii,ci);
            else temp=quart_ptr->tent;
            if temp < 0 then do;
                        sign=-1;
                        temp=-temp;
                        end;
            else sign=1;
            i=12;
         comp:
            rem=temp;
            temp=divide(temp,10,35,0);
            rem=rem-10*temp;
            rem=rem+48;          /* ASCII conversion */
            c11(i)=substr(false,4,1);
```

```
            i=i-1;
            if temp^=0 then go to comp;
            if sign=-1 then do;
                       cl1(i)="-";
                       i=i-1;
                       end;
            pseudo_length=12-i;
            input_or_return_string=substr(cl12,i+1,pseudo_leng
th);
            if ii<upper_bound then go to loop_7;
            revert fixedoverflow;
            return;


/*****************************************************"******
**********/


get_refno:
            ii=0;
       loop_0:
            ii=ii+1;
            str_ptr=d_or_q_ptr(ii);
            allocate quart set (quart_ptr);
            cent=data_type;
            pent=seg_ptr;
            allocate rem set (con_ptr);
            return_ptr(ii)=quart_ptr;
            cl12=substr(input_or_return_string,1,pseudo_length

            if cl1(1)="-" then do;
                       sloc=2;
                       sign=-1;
                       end;
            else if cl1(1)="+" then do;
                       sloc=2;
                       sign=1;
                       end;
            else do;
                       sloc=1;
                       sign=1;
                       end;
            temp=0;
            rem=0;
            i=sloc-1;
        loop:
            i=i+1;
            substr(false,4,1)=cl1(i);
            if (rem<48|em>57) then do;
                       dsm_args.return_ptr(ii)=addr(mt_quart);
                       mtpent=ddptr;
                       free quart_ptr->quart;
                       go to end_of_gr;
                       end;
```

```
            temp=temp+(rem-48)*pot(pseudo_length-i);
         end_of_gr:
            if i<pseudo_length then go to loop;
            tent=temp*sign;
            if ii < upper_bound then go to loop_0;
            return;

/*****************************************************************
**********/


new_data_type:
            ii=0;
       llllooop:
            ii=ii+1;
            call hcs_$fs_get_path_name(seg_ptr,dir,grech,ename
,code);
            call area_(1024,seg_ptr);
            allocate type set (type_ptr) in (data_seg);
            data_type=ename;
            dsm_name="dsm_integer";
            if ii<upper_bound then go to llllooop;
            return;

/*****************************************************************
*********/


end dsm_integer;
```

dsm_chain.pl1 06/01/70 1830.8 edt Mon

```
dsm_chain: proc (dsm_args_ptr);

/*************************************************************
*********/

dcl data_seg area (1024) based (seg_ptr);

dcl 1 dsm_args based (dsm_args_ptr),
          2 upper_bound fixed bin (17),
          2 ci fixed bin (17),
          2 op char (1),
          2 di char (1),
          2 c char (1),
          2 ddptr ptr,
          2 d_or_q_ptr (upper_bound) ptr,
          2 return_ptr (upper_bound) ptr;

dcl 1 quart based (quart_ptr),
          2 id fixed bin (35) initial (0),
          2 length fixed bin (35) initial (1),
          2 order fixed bin (35) initial (1),
          2 cent char (32) initial (data_type),
          2 pent ptr initial (seg_ptr),
          2 tent fixed bin (35);

dcl 1 quart_proper based (quart_ptr),
          2 qid fixed bin (35),
          2 qlength fixed bin (35),
          2 qorder fixed bin (35),
          2 qcent (qorder) char (32),
          2 qpent (qorder) ptr,
          2 qtent (qlength,qorder) fixed bin (35);

dcl 1 pseudo_string_array based (str_ptr),
          2 pseudo_length fixed bin (35),
          2 input_or_return_string char (mlen_int);

dcl 1 type based (type_ptr),
          2 dsm_name char (16) initial ("dsm_chain"),
          2 data_type char (32),
          2 num_entries fixed bin (35),
          2 successor_chain_head offset (data_seg),
          2 max_length fixed bin (35),
          2 first_refno fixed bin (35);
```

```
begin_dsm_chain:

   exp=dsm_args.e;
   dori=dsm_args.di;
   opcode=dsm_args.op;
   seg_ptr=ddptr;
   type_ptr=first_off;


     if opcode="D" then   /* input is refnos */
       if dori=" " then go to get_datum;
         else if dori="D" then go to delete_by_refno;
           else go to op_error;

     else if opcode="R" then     /* data strings are input */

       if dori=" " then go to get_refno;
       else if dori ="D" then go to delete_by_datum;
       else if dori="I" then go to insert;
         else go to op_error;

       else if opcode="S" then
         if dori =" " then go to successor;
         else go to op_error;


     else if opcode="I" then
       if (dori=" " & exp=" ") then go to new_data_type;


      op_error: do ii = 1 to upper_bound;
                  return_ptr(ii)=addr(mt_quart);
                   mtcent=data_type;
                   mtpent=seg_ptr;
                  end;
             return;

/*****************************************************************
********/


new_data_type:
           do ii = 1 to upper_bound;
             call area_(1024,seg_ptr);
             call hcs_$fs_get_path_name(seg_ptr,dirname,grech
,ename,code);
             allocate type set (type_ptr) in (data_seg);
             num_entries=0;
             call ioa_("what is max length for character stri
ngs for ~a>~a ?~/",substr(dirname,1,grech),ename);
             call read_list_(grech);
```

```
      end;
    return;


/*************************************************************
**********/


successor:
           do ii = 1 to upper_bound;
                     return_label=successor_return;
                     go to refno_prologue;
                   successor_return:
                     return_label=s_;
                     go to quart_alloc;
                   s_:
                     if found=-1 then length=0;
                                 else if ptr=null then tent=suc
cessor_chain_head->refno;
                                                 else if ptr->succ=nu
ll then length=0;
                     else tent=ptr->succ->refno;
                     end;
           return;


/*************************************************************
**********/


delete_by_refno:
           do ii = 1 to upper_bound;
                     return_label=delete_by_refno_return;
                     go to refno_prologue;
                   delete_by_refno_return:
                     if found^=1 then do;
                                 return_ptr(ii)=addr(mt_quart);

                                 mtcent=data_type;
                                 mtpent=seg_ptr;
                                 end;
                       else do;
                                 return_label=rd;
                                 go to string_alloc;
                               rd:
                                 pseudo_length=ptr->string_leng
th;
                                 input_or_return_string=substr(
ptr->string,1,ptr->string_length);
                                 if pred=null then successor_ch
ain_head=ptr->succ;
                                 else pred->succ=ptr->succ;
                                 free ptr->item;
                                 num_entries=num_entries-1;
```

```
                                        iptr->string=substr(
input_or_return_string,1,pseudo_length);
                                        iptr->string_length=
pseudo_length;
                                        end;
                                end;
                        iend: end;
            return;


/*************************************************************
**********/


quart_alloc:
            allocate quart set (quart_ptr);
            return_ptr(ii)=quart_ptr;
            pent=seg_ptr;
            cent=data_type;
            go to return_label;

string_alloc:
            mlen_int=max_length;
            allocate pseudo_string_array set (str_ptr);
            return_ptr(ii)=str_ptr;
            go to return_label;


/*************************************************************
**********/


refno_prologue:
            if exp="F" then quart_ptr=d_or_q_ptr(1);
            else quart_ptr=d_or_q_ptr(ii);

         search_by_refno:
            found=0;
            if successor_chain_head=null() then do;
                        found=-1;
                        go to return_label;
                        end;
            ptr=successor_chain_head;
            pred=null;
            if exp="F" then presented_refno=qtent(ii,ci);
            else presented_refno=tent;
         refno_loop:
            if presented_refno>ptr->refno then go to refno_fut
z;
            if presented_refno < ptr->refno then do;
                        ptr=pred;
                        go to return_label;
                        end;
            found=1;
```

```
            go to return_label;
         refno_futz:
            pred=ptr;
            glich=ptr->succ;
            if glich=nullo then do;
                     go to return_label;
                     end;
            ptr=glich;
            go to refno_loop;


/*****************************************************************
**********/

datum_prologue:
            str_ptr=d_or_q_ptr(ii);

         search_by_datum:
            found=0;
            if successor_chain_head=nullo then do;
                     found=-1;
                     go to return_label;
                     end;
            ptr=successor_chain_head;
            pred=null;
            presented_string=substr(input_or_return_string,1,p
seudo_length);
         datum_loop:
            if presented_string^=substr(ptr->string,1,ptr->str
ing_length) then go to datum_futz;
            found=1;
            go to return_label;
         datum_futz:
            pred=ptr;
            glich=ptr->succ;
            if glich=nullo then  do;
                     go to return_label;
                     end;
            ptr=glich;
            go to datum_loop;


/*****************************************************************
**********/


end dsm_chain;
EOF
```

dsm_table.pl1 06/01/70 1758.3 edt Mon

```
dsm_table: proc (dsm_args_ptr);

/*************************************************************
*********/

dcl data_seg area (area_size) based (seg_ptr);

dcl 1 dsm_args based (dsm_args_ptr),
           2 upper_bound fixed bin (17),
           2 ci fixed bin (17),
           2 op char (1),
           2 di char (1),
           2 e char (1),
           2 ddptr ptr,
           2 d_or_q_ptr (upper_bound) ptr,
           2 return_ptr (upper_bound) ptr;

dcl 1 quart based (quart_ptr),
           2 id fixed bin (35) initial (0),
           2 length fixed bin (35) initial (1),
           2 order fixed bin (35) initial (1),
           2 cent char (32) initial (data_type),
           2 pent ptr initial (seg_ptr),
           2 tent fixed bin (35);

  dcl 1 quart_proper based (quart_ptr),
           2 qid fixed bin (35),
           2 qlength fixed bin (35),
           2 qorder fixed bin (35),
           2 qcent (qorder) char (32),
           2 qpent (qorder) ptr,
           2 qtent (qlength,qorder) fixed bin (35);
dcl 1 pseudo_string_array based (str_ptr),
           2 pseudo_length fixed bin (35),
           2 input_or_return_string char (max_length);

dcl 1 type based (type_ptr),
           2 dsm_name char (16),
           2 data_type char (32),
           2 tab_off offset (data_seg),
           2 num_entries fixed bin (35) initial (in_len),
           2 max_length fixed bin (35) initial (in_max);


dcl 1 table (in_len) based (tab_off),
           2 string_length fixed bin (35),
           2 string char (max_length);
```

```
dcl 1  mt_quart static,
           2 mtid fixed bin (35) initial (0),
           2 mtlength fixed bin (35) initial (0),
           2 mtorder fixed bin (35) initial (1),
           2 mtcent char (32),
           2 mtpent ptr;

dcl presented_string char (168) varying static;
declare exp char(1) varying;
dcl (dirname char (168), ename char (32), grech fixed bin (3
5)) static;

dcl (seg_ptr,type_ptr,str_ptr,quart_ptr) ptr static;

dcl code fixed bin (17);

dcl (addr,divide,substr,max) builtin;

dcl dsm_args_ptr ptr;

dcl first_off offset (data_seg) based (seg_ptr);

dcl (,ii,tab_len,in_len,in_max,jj,area_size,char_len) fixed
 bin (35) static;

dcl return_label label (r_,s_,d_);

/*************************************************************
**********/


begin_dsm_table:
    seg_ptr=ddptr;
    type_ptr=first_off;
    if substr(op,1,1)="D" then go to get_datum;   /* input i
s refnos */
    else if substr(op,1,1)="R" then go to get_refno;     /*
data strings are input */
      else if substr(op,1,1)="S" then go to successor;
    else if substr(op,1,1)="N" then go to new_data_type;
     else do ii = 1 to upper_bound;
                return_ptr(ii)=addr(mt_quart);
                 mtcent="";
                 mtpent=ddptr;
                end;
            return;


/*************************************************************
********/
```

```
new_data_type:
          do ii = 1 to upper_bound;
            call hcs_$fs_get_path_name(seg_ptr,dirname,grech
,ename,code);
            call ioa_("what is the number of table entries in
data type ^a>^a ?^/",substr(dirname,1,grech),ename);
            call read_list_(in_len);
            call ioa_("what is max length for character strin
s in ^a>^a ? ^/",substr(dirname,1,grech),ename);
            call read_list_(in_max);
            area_size=max(1024,14+in_len*(1+divide(in_max,4,35
,0)+1));
            call area_(area_size,seg_ptr);
            allocate type# set (type_ptr) in (data_seg);
            allocate table set (tab_off) in (data_seg);
            num_entries=0;
             data_type=ename;
             dsm_name="dsm_table";
             end;
         return;


/*************************************************************
************/


get_datum:
        do ii = 1 to upper_bound;
           if exp="E" then m=d_or_q_ptr(1)->qtent(ii,ci);
           else m=d_or_q_ptr(ii)->tent;
           if ((m<=num_entries) & (table(m).string_length^=-1
)) then do;
            return_label=d_;
            go to string_alloc;
           d_:
            pseudo_length=table(m).string_length;
             input_or_return_string=substr(table(m).string,1,
pseudo_length);
             if di="D" then table(m).string_length=-1;
            end;
            else do;
               dsm_args.return_ptr(ii)=addr(mt_quart);
                mtcent=data_type;
                mtpent=seg_ptr;
                end;
      end;
    return;


/************************************/
```

```
get_refno:
        do ii = 1 to upper_bound;
                        str_ptr=d_or_q_ptr(ii);
                        presented_string=substr(input_or_return_
string,1,pseudo_length);
                        do jj = 1 to num_entries;
                                        if table(jj).string_length^=-1
 then
                                                        if presented_string=
substr(table(jj).string,1,table(jj).string_length) then go t
o grq;
                        end;
                        if di="I" then go to insert;
                        else do;
                                        dsm_args.return_ptr(ii)=addr(m
t_quart);
                                        mtcent=data_type;
                                        mtpent=seg_ptr;
                                        go to i_end;
                                        end;
                        insert:
                                        num_entries=num_entries+1;
                                        table(num_entries).string_leng
th=pseudo_length;
                                        table(num_entries).string=subs
tr(input_or_return_string,1,pseudo_length);
                                        jj=num_entries;
                    grq:
                        return_label=r_;
                        go to quart_alloc;
                    r_:
                        tent=jj;
                        if di="D" then table(jj).string_length=-
1;
                   i_end: end;
    return;


/*************************************************************
**********/


successor:
            do ii = 1 to upper_bound;
                        if exp="F" then m=d_or_q_ptr(1)->qtent(i
i,ci);
                       else m=d_or_q_ptr(ii)->tent;
                       return_label=s_;
                       go to quart_alloc;
                     s_:
                       if m>num_entries | m<0 then do;
                                return_ptr(ii)=addr(mt_quart);
```

```
                                free quart_ptr->quart;
                                mtcent=data_type;
                                mtpent=seg_ptr;
                                go to succ_end;
                                end;
                        sloop:
                          m=m+1;
                          if m>num_entries then length=0;
                          if table(m).string_length=-1 then go to
sloop;
                          tent=m;
             succ_end: end;
            return;


/*****************************************************************
**********/


quart_alloc:
            allocate quart set (quart_ptr);
            pent=seg_ptr;
            cent=data_type;
            return_ptr(ii)=quart_ptr;
            go to return_label;

string_alloc:
            allocate pseudo_string_array set (str_ptr);
            return_ptr(ii)=str_ptr;
            go to return_label;


/*****************************************************************
**********/


end dsm_table;
```

rsm_wq.pl1 06/03/7u 1014.5 edt Wed

```
rsm_wq:   procedure(arg_ptr);
          declare 1 args  based (arg_ptr),
              2 upper_bound fixed bin(35),
              2 op character(1),
              2 q1_ptr (limit refer (upper_bound)) pointer,
              2 r1_ptr (limit refer (upper_bound)) pointer,
 /*Not used here*/
              2 q2_ptr (limit refer (upper_bound)) pointer,
              2 r2_ptr (limit refer (upper_bound)) pointer,
  /*Not used here*/
              2 equiv_ptr (limit refer (upper_bound)) pointe
r,  /*Not used here*/
              2 return_arg (limit refer (upper_bound)) point
er;
          declare 1 quart based (lptr),
              2 ID fixed bin(35) initial(0),
              2 length fixed bin(35),
              2 order fixed bin(35),
              2 cent(alloc_order refer (order)) char(32)
              2 pent(alloc_order refer (order)) pointer,
              2 tent(alloc_length refer (length),alloc_order
 refer (order)) fixed bin (35);
          declare 1 ent_vector based (rptr),
              2 width fixed bin(35),
              2 names(alloc_order refer (width)) char(32);
          declare 1 int_sort_quart,
              2 sid fixed bin(35) initial(0),
              2 slength fixed bin(35) initial(0),
              2 sorder fixed bin (35) initial(20),
              2 scent(sorder) char(32);
          declare 1 feht based (eq_ptr),
              2 num fixed bin(35),
              2 newcent (1 refer (num)) char(32);
          declare return_place label (major_loop,in_UID_op,i
n_P_op,in_C_op_1,in_C_op_2);
          declare op_label label (UID_operate,S_operate,P_op
erate,X_operate,Z_operate,M_operate,
                                  E_operate,Q_operate,O_oper
ate,C_operate);
          declare q_alloc_r label (q_alloc_1,q_alloc_2,q_all
oc_3,q_alloc_4,q_alloc_5,q_alloc_6,q_alloc_7,q_alloc_8);


          declare (gen_purpose_ptr,new_ptr,free_ptr,eq_ptr,t
ptr,cptr) pointer;
          declare (lptr,rptr,temp_ptr,temp2_ptr,temp_lptr,te
mp_rptr) pointer;
          declare (null,max) builtin;
```

```
          declare corres(slength) fixed bin(35) based (gen_p
urpose_ptr);
          declare token(slength) fixed bin(35) based (gen_pu
rpose_ptr);
          declare int_cent(20) char(32);
          declare alloc_cent(20) char(32);
          declare cent_flag(20) fixed initial((20) 0);


          declare temp fixed bin(35);
          declare (n_sw,no_sort_sw) fixed bin (1);
          declare (step_sw) fixed bin(1) initial(0b);
          declare (lplace,rplace,oplace,tplace,alloc_order,a
lloc_length,entry_count) fixed bin(17);
          declare (slength,rrow,rcol,lrow,lcol,nc,nr,c1,lt,r
t,free_len,free_ord) fixed bin(17);
          declare (ii,jj,kk,ll,mm,i,j,limit,ocount,place,chu
nk) fixed bin(17);
          declare (llength,rlength,lorder,rorder,l_new_order
) fixed bin(17);



          if args.op="U"|args.op="I"|args.op="D" then op_lab
el=UID_operate;
          if args.op="P" then op_label=P_operate;
          if args.op="S" then op_label=S_operate;
          if args.op="X" then op_label=X_operate;
          if args.op="O" then op_label=O_operate;
          if args.op="C" then op_label=C_operate;
          if args.op="E" then op_label=E_operate;
          if args.op="M" then op_label=M_operate;
          if args.op="Z" then op_label=Z_operate;
          if args.op="Q" then op_label=Q_operate;


          do kk=1 to args.upper_bound;
          if args.op="U"|args.op="I"|args.op="D" then go to
no_set_ptrs;
          lptr=args(kk).q1_ptr;
          rptr=args(kk).q2_ptr;
no_set_ptrs:
          return_place=major_loop;
          go to op_label;


quart_alloc: allocate quart set (temp_ptr);
          go to q_alloc_r;


E_operate: alloc_order=order;
          allocate ent_vector set(args(kk).return_arg);
          do i=1 to alloc_order;
              args(kk).return_arg->names(i)=cent(i);
```

```
            end;
            go to return_place;


Z_operate: args(kk).return_arg=lptr;
            do i=1 to order;
                cent(i)=null;
            end;
            go to return_place;


M_operate: args(kk).return_arg=lptr;
            if width¬=order then do;
                call GS_error$pr(103,"M",lptr,rptr);
                args.return_arg(kk)=null;
                go to return_place;
            end;
            do i=1 to order;
                if names(i)¬="&" then cent(i)=names(i);
            end;
            go to return_place;


O_operate: if width¬=order then do;
                call GS_error$pr(103,"O",lptr,rptr);
null_answer:    args.return_arg(kk)=null;
                go to return_place;
            end;
            slength=order;
            alloc_order=slength;
            allocate ent_vector set(cptr);
            allocate token set(tptr);
            jj=1;
            do mm=1 to order;
                if names(mm)="&" then cptr->names(mm)=cent(mm)
;
                else cptr->names(mm)=names(mm);
                if names(mm)¬=" " then do;
                    tptr->token(jj)=mm;
                    jj=jj+1;
                end;
            end;
            if jj>1 then alloc_order=jj-1;
            else go to null_answer;
            alloc_length=length;
            q_alloc_r=q_alloc_7;
            go to quart_alloc;
q_alloc_7:tplace=1;
            do mm=1 to alloc_order;
                temp_ptr->cent(mm)=cptr->names(tptr->token(mm)
);
                temp_ptr->pent(mm)=null;
            end;
            do jj=1 to length;
                p_sw=0b;
```

```
            if tent(lplace,oplace)>rptr->tent(rplace,oplac
e) then do;
                if args.op="U" then do;
                    do ocount=1 to alloc_order;
                        temp_ptr->tent(tplace,ocount)=rptr
->tent(rplace,ocount);
                    end;
                    tplace=tplace+1;
                end;
                if rptr->length>rplace then do;
                    rplace=rplace+1;
                    go to neworder;
                end;
                else finish_left: do place=lplace to lengt
h while(args.op="U"|args.op="D");
                    do ocount=1 to alloc_order;
                        temp_ptr->tent(tplace,ocount)=tent
(lplace,ocount);
                    end;
                    tplace=tplace+1;
                end;
                go to stop_action;
            end;
            else do;
                if args.op="U"|args.op="D" then do;
                    do ocount=1 to alloc_order;
                        temp_ptr->tent(tplace,ocount)=tent
(lplace,ocount);
                    end;
                    tplace=tplace+1;
                end;
                if length>lplace then do;
                    lplace=lplace+1;
                    go to neworder;
                end;
                else finish_right: do place=rplace to rptr
->length while(args.op="U");
                    do ocount=1 to alloc_order;
                        temp_ptr->tent(tplace,ocount)=rptr
->tent(rplace,ocount);
                    end;
                    tplace=tplace+1;
                end;
                go to stop_action;
            end;
        end;
        else do;
            if order>oplace then do;
                oplace=oplace+1;
                go to check;
            end;
            if args.op="D" then go to nocopy_right;
            do ocount=1 to alloc_order;
                temp_ptr->tent(tplace,ocount)=tent(lplace,
ocount);
```

```
                end;
                tplace=tplace+1;
nocopy_right:   if length¬>lplace then do;
                    if rptr->length>rplace then do;
                        rplace=rplace+1;
                        go to finish_right;
                    end;
                    else go to stop_action;
                end;
                else do;
                    lplace=lplace+1;
                    if rptr->length¬>rplace then go to finish_
left;
                    rplace=rplace+1;
                    go to neworder;
                end;
            end;
stop_action: oplace=alloc_length;
            alloc_length=tplace-1;
            if alloc_length=oplace then go to no_new_copy;
            temp2_ptr=temp_ptr;
            q_alloc_r=q_alloc_2;
            go to quart_alloc;
q_alloc_2:  do jj=1 to alloc_order;
                temp_ptr->cent(jj)=cent(jj);
                temp_ptr->pent(jj)=pent(jj);
                do mm=1 to alloc_length;
                    temp_ptr->tent(mm,jj)=temp2_ptr->tent(mm,j
j);
                end;
            end;
            alloc_length=oplace;
            free temp2_ptr->quart;
no_new_copy:args(kk).return_arg=temp_ptr;
            go to return_place;


P_operate:  if rptr=null then go to get_first_element;
            if rptr->length=0 then go to get_first_element;
            alloc_order=order;
            slength=alloc_order;
            if rptr->order¬=alloc_order then do;
                args.return_arg(kk)=null;
                call GS_error$pr(101,op,lptr,rptr);
                go to return_place;
            end;
            allocate corres set (cptr);
            do ii=1 to alloc_order;
                do jj=1 to alloc_order;
                    if rptr->cent(jj)=cent(ii) then do;
                        cptr->corres(ii)=rptr->tent(1,jj);
                        go to next_step_on;
                    end;
                end;
```

```
C_operate:
                eq_ptr=equiv_ptr(kk);
                llength=lptr->length;
                rlength=rptr->length;
                rorder=rptr->order;
                lorder=lptr->order;
                if (lorder+rorder)¬=feht.num then
                    call GS_error$p(105,"C",eq_ptr,null);


/* 1. put new bindings in int_cent */

           do i = 1 to lorder;
                if newcent(i)="&" then int_cent(i)=lptr->cent(
i);
                else int_cent(i)=newcent(i);
                end;
           do j = 1 to rorder;
                if newcent(lorder+j)="&" then int_cent(lorder+
j)=rptr->cent(j);
                else int_cent(lorder+j)=newcent(lorder+j);
                end;

/* 2. sort left quart on cents in domain of lambda (see thes
is) */

lsort:     k=1;
           sorder=lorder;
           do i=1 to lorder;
                if cent_flag(i)=1|int_cent(i)=" " then go to l
nofill;
                do m=lorder+1 to lorder+rorder;
                    if int_cent(m)=int_cent(i) then do;
                        cent_flag(i)=1;
                        scent(k)=lptr->cent(i);
                        k=k+1;
                        go to jtest;
                    end;
                end;
                go to lnofill;
jtest:          do j=i+1 to lorder;
                    if int_cent(i)=int_cent(j) then do;
                        cent_flag(j)=1;
                        scent(k)=lptr->cent(j);
                        k=k+1;
                    end;
                end;
lnofill:   end;
contin:    do i = 1 to (lorder-1);
```

```
                if cent_flag(i) = 1|int_cent(i)=" " then go to
 liend;
                cent_flag(i)=1;
                scent(k)=lptr->cent(i);
                k=k+1;
                do j = (i+1) to lorder;
                    if int_cent(i)=int_cent(j) then do;
                        cent_flag(j)=1;
                        scent(k)=lptr->cent(j);
                        k=k+1;
                    end;
liend:          end;
            do i=1 to lorder;
                if cent_flag(i)=0 then do;
                    cent_flag(i)=1;
                    scent(k)=lptr->cent(i);
                    k=k+1;
                end;
            end;
            rptr=addr(int_sort_quart);
            return_place=in_C_op_1;
            go to S_operate;
in_C_op_1: temp_ptr=return_arg(kk);
            temp_lptr=temp_ptr;
            rptr=q2_ptr(kk);
            do i = 1 to lorder;
                do j = 1 to lorder;
                    if temp_ptr->cent(i)=lptr->cent(j) then
                        temp_ptr->cent(i)=int_cent(i);
                    end;
                end;

/* 3. check right quart for need to sort */
rsort:
            k=1;
            sorder=rorder;
            rptr=q2_ptr(kk);
            do i = lorder+1 to lorder+rorder;
                if cent_flag(i)=1|int_cent(i)=" " then go to r
nofill;
                do j=1 to lorder;
                    if int_cent(i)=int_cent(j) then do;
                        cent_flag(i)=1;
                        scent(k)=rptr->cent(i-lorder);
                        k=k+1;
                        go to mtest;
                    end;
                end;
                go to rnofill;
```

```
mtest:          do m=i+1 to rorder+lorder;
                    if int_cent(i)=int_cent(m) then do;
                        cent_flag(m)=1;
                        scent(k)=rptr->cent(m-lorder);
                        k=k+1;
                    end;
                end;
rnofill:    end;


            do i = (lorder+1) to (lorder+rorder);
                if cent_flag(i)=1|int_cent(i)=" " then go to r
iend;
                cent_flag(i)=1;
                scent(k)=rptr->cent(i-lorder);
                k=k+1;
                do m=i+1 to lorder+rorder;
                    if int_cent(m)=int_cent(i) then do;
                        cent_flag(m)=1;
                        scent(k)=rptr->cent(m-lorder);
                        k=k+1;
                    end;
                end;
riend:      end;
            do i=1+lorder to lorder+rorder;
                if cent_flag(i)=0 then do;
                    cent_flag(i)=1;
                    scent(k)=rptr->cent(i-lorder);
                    k=k+1;
                end;
            end;
            lptr=rptr;

            rptr=addr(int_sort_quart);
            return_place=in_C_op_2;
            go to S_operate;
in_C_op_2:  temp_ptr=return_arg(kk);
            rptr=q2_ptr(kk);
            lptr=temp_lptr;
            do i = 1 to rorder;
                do j = (lorder+1) to (lorder+rorder);
                    if temp_ptr->cent(i)=rptr->cent(j-lorder)
then
                        rptr->cent(j-lorder) = int_cent(j);
                end;
            end;

                alloc_order=1;
                do i = 1 to (lorder-1);
                    if lptr->cent(i)=lptr->cent(i+1) then go t
o aliend;
                    else if lptr->cent(i)¬=" " then do;
                      alloc_cent(alloc_order)=lptr->cent(i);
                      alloc_order=alloc_order+1;
```

```
                    end;
                  aliend: end;
                if lptr->cent(lorder-1)¬=lptr->cent(lorder) th
en
                  if lptr->cent(lorder)¬=" " then do;
                    alloc_cent(alloc_order)=lptr->cent(lorder)
;
                    alloc_order=alloc_order+1;
                    end;
                l_new_order=alloc_order-1;

                do i = 1 to (rorder-1);
                    do j = 1 to l_new_order;
                      if alloc_cent(j)=rptr->cent(i) then go t
o ariend;
                      end;
                    if rptr->cent(i)=rptr->cent(i+1) then go t
o ariend;
                    else if rptr->cent(i)¬=" " then do;
                      alloc_cent(alloc_order)=rptr->cent(i);
                      alloc_order=alloc_order+1;
                      end;
                  ariend: end;
                if rptr->cent(rorder)¬=rptr->cent(rorder-1) th
en
                  if rptr->cent(rorder)¬=" " then do;
                    alloc_cent(alloc_order)=rptr->cent(rorder)
;
                    alloc_order=alloc_order+1;
                    end;
                alloc_order=alloc_order-1;
                alloc_length=rlength*llength;
                temp_rptr=temp_ptr;
                q_alloc_r=q_alloc_8;
                go to quart_alloc;
q_alloc_8:              new_ptr=temp_ptr;
                do i = 1 to alloc_order;
                    new_ptr->cent(i)=alloc_cent(i);
                    end;
                do i = 1 to lorder;
                    if rptr->cent(1)=lptr->cent(i) then c1 =i;

                    end;
begin_compose:

          lrow,rrow,nr=1;
        row_incr:
          if lrow>llength|rrow>rlength then do;
              nr=nr-1;
              go to return_sequence;
              end;
          if lptr->tent(lrow,c1)<rptr->tent(rrow,1) then do;
```

```
                lrow=lrow+1;
                go to row_incr;
                end;
            if lptr->tent(lrow,c1)>rptr->tent(rrow,1) then do;

                rrow=rrow+1;
                go to row_incr;
                end;


one_compose:
            lcol,rcol,nc=1;
          lr_comp:
            if lptr->cent(lcol)¬=rptr->cent(rcol)|lptr->cent(l
col)=" " then do;
                lcol_incr_0:
                lcol=lcol+1;
                if lcol>lorder then do;
                    new_ptr->tent(nr,nc)=lptr->tent(lrow,lcol-
1);
                    nc=nc+1;
                    go to right_finish;
                    end;
                else if lptr->cent(lcol)=" " then go to lcol_i
ncr_0;
                else if lptr->cent(lcol)=lptr->cent(lcol-1) th
en
                    if lptr->tent(lrow,lcol)=lptr->tent(lrow,l
col-1) then
                      go to lcol_incr_0;
                    else do;
                      lrow=lrow+1;
                      go to row_incr;
                      end;
                else do;
                    new_ptr->tent(nr,nc)=lptr->tent(lrow,lcol-
1);
                    nc=nc+1;
                    go to lr_comp;
                    end;
                end;
            else if lcol>=lorder then do;
                lcol=lcol+1;
                if lptr->tent(lrow,lcol-1)¬=rptr->tent(rrow,rc
ol) then go to right_finish;
                else do;
                    new_ptr->tent(nr,nc)=lptr->tent(lrow,lcol-
1);
                    nc=nc+1;
                    go to right_finish;
                    end;
                end;
```

```
          else if lptr->cent(lcol)=lptr->cent(lcol+1) then
               if lptr->tent(lrow,lcol)¬=lptr->tent(lrow,lcol
+1) then do;
                    lrow=lrow+1;
                    go to row_incr;
                    end;
               else do;
                    lcol=lcol+1;
                    go to lr_comp;
                    end;
          else if rcol>=rorder then do;
               rcol=rcol+1;
               if lptr->tent(lrow,lcol)¬=rptr->tent(rrow,rcol
-1) then go to left_finish;
               else do;
                    new_ptr->tent(nr,nc)=rptr->tent(rrow,rcol-
1);
                    nc=nc+1;
                    go to left_finish;
                    end;
               end;
          else if lptr->tent(lrow,lcol)¬=rptr->tent(rrow,rco
l) then do;
               if lptr->tent(lrow,c1) < rptr->tent(rrow,1) th
en rrow=rrow+1;
               else lrow=lrow+1;
               go to row_incr;
               end;
          else do;
               new_ptr->tent(nr,nc)=lptr->tent(lrow,lcol);
               lcol=lcol+1;
               rcol=rcol+1;
               nc=nc+1;
               go to lr_comp;
               end;


right_finish:
          if rcol>rorder then do;
               nr=nr+1;
               if lptr->tent(lrow,c1) < rptr->tent(rrow,1) th
en rrow=rrow+1;
               else lrow=lrow+1;
               go to row_incr;
               end;
          else if rptr->cent(rcol)=" " then do;
               rcol=rcol+1;
               go to right_finish;
          end;
          else if rptr->cent(rcol)=rptr->cent(rcol+1) then
               if rptr->tent(rrow,rcol)=rptr->tent(rrow,rcol+
1) then do;
                    rcol=rcol+1;
                    go to right_finish;
```

```
                end;
            else do;
                if lptr->tent(lrow,c1) < rptr->tent(rrow,1
) then rrow=rrow+1;
                else lrow=lrow+1;
                go to row_incr;
                end;
        else do;
            new_ptr->tent(nr,nc)=rptr->tent(rrow,rcol);
            nc=nc+1;
            rcol=rcol+1;
            go to right_finish;
            end;



left_finish:
        if lcol>lorder then do;
            nr=nr+1;
            if lptr->tent(lrow,c1) < rptr->tent(rrow,1) th
en rrow=rrow+1;
            else lrow=lrow+1;
            go to row_incr;
            end;
        else if lptr->cent(lcol)=" " then do;
            lcol=lcol+1;
            go to left_finish;
        end;
        else if lptr->cent(lcol)=lptr->cent(lcol+1) then
            if lptr->tent(lrow,lcol)=lptr->tent(lrow,lcol+
1) then do;
                lcol=lcol+1;
                go to left_finish;
                end;
            else do;
                lrow=lrow+1;
                go to row_incr;
                end;
        else do;
            new_ptr->tent(nr,nc)=lptr->tent(lrow,lcol);
            nc=nc+1;
            lcol=lcol+1;
            go to left_finish;
            end;



        tplace=nr-1;
        free temp_lptr->quart;
        free temp_rptr->quart;
        return_place=major_loop;
        go to stop_action;
```

```
S_operate: slength=length;
           if rptr=null then go to skip_1;
           if order¬=rptr->order then do;
               args(kk).return_arg=null;
               call GS_error$pr(101,op,lptr,rptr);
               go to return_place;
           end;
/*Error here is unequal orders in sort*/
skip_1:    alloc_length=length;
           alloc_order=order;
           no_sort_sw=1b;
           allocate token set (tptr);
           allocate corres set (cptr);
           if rptr=null then do;
               do ii=1 to alloc_order;
                   corres(ii)=ii;
               end;
               go to skip_2;
           end;
           do ii=1 to alloc_order;
               do jj=1 to alloc_order;
                   if rptr->cent(ii)=cent(jj) then do;
                       cptr->corres(ii)=jj;
                       if ii¬=jj then no_sort_sw=0b;
                       go to one_corres_down;
                   end;
               end;
           args(kk).return_arg=null;
           free cptr->corres;
           free tptr->token;
           call GS_error$pr(102,op,lptr,rptr);
           go to return_place;
/*Error here is inconsistant entries in the two cent tuples*
/
one_corres_down: end;
    /*Set up the tptr->tokens*/
skip_2:    do ii=1 to alloc_length;
               tptr->token(ii)=ii;
           end;
    /*Main sort*/
           if no_sort_sw then go to sort_done;
           do ii=1 to alloc_length;
               do jj=1 to (alloc_length-1);
                   n_sw=0;
                   do mm=1 to alloc_order;
                       if tent(tptr->token(jj),cptr->corres(m
m))>=tent(tptr->token(jj+1),cptr->corres(mm)) then do;
                           temp=tptr->token(jj+1);
                           tptr->token(jj+1)=tptr->token(jj);

                           tptr->token(jj)=temp;
                           n_sw=1;
                           go to pop_extra_loop;
```

```
                        end;
                    end;
pop_extra_loop: end;
                if n_sw=0 then go to sort_done;
            end;
sort_done: q_alloc_r=q_alloc_4;
            go to quart_alloc;
q_alloc_4: do ii=1 to alloc_order;
                temp_ptr->cent(ii)=cent(cptr->corres(ii));
                temp_ptr->pent(ii)=pent(cptr->corres(ii));
            #    do jj=1 to alloc_length;
                    temp_ptr->tent(jj,ii)=tent(tptr->token(jj)
,cptr->corres(ii));
                end;
            end;
            free tptr->token;
            free cptr->corres;
            args(kk).return_arg=temp_ptr;
            go to return_place;


major_loop: end;
            return;
error:      end;
```

```
rsm_q:    procedure(argptr);
          declare 1 args based (argptr),
              2 bound fixed bin(35),
              2 op char(1),
              2 q1ptr (limit refer (bound)) pointer,
              2 r1ptr (limit refer (bound)) pointer,
              2 q2ptr (limit refer (bound)) pointer,
              2 r2ptr (limit refer (bound)) pointer,
              2 eptr (limit refer (bound)) pointer,
              2 retptr (limit refer (bound)) pointer;
          declare mgr$cq entry external returns(pointer);
          declare 1 quart based (quart_ptr),
              2 ID fixed bin(35) initial(0),
              2 length fixed bin(35),
              2 order fixed bin(35),
              2 cent(alloc_order refer (order)) char(32),
              2 pent(alloc_order refer (order)) pointer,
              2 tent(alloc_length refer (length),alloc_order
 refer (order)) fixed bin(35);
          declare oset bit(18) based (gen_purpose_ptr);
          declare new_area area (area_size) based (retptr(kk
));
          declare old_area area (area_size) based (r1ptr(kk)
);
          declare old_area2 area (area_size) based (r2ptr(kk
));
          declare 1 header based (hdr_ptr),
              2 rsm_name char(16),
              2 name_r char(32),
              2 quart_location offset (old_area);
          declare (alloc_length,alloc_order,area_size,jj,kk,
limit,mm)fixed bin(17);
          declare (argptr,gen_purpose_ptr,hdr_ptr,new_args_p
tr,quart_ptr,use) pointer;



          if op="N" then go to no_call;
          limit=bound;
          allocate args set (new_args_ptr);
          new_args_ptr->op=op;
          if op="R" then do;
              do kk=1 to limit;
                  if q1ptr(kk)¬=null then new_args_ptr->retp
tr(kk)=q1ptr(kk);
                  else if addrel(r1ptr(kk),r1ptr(kk)->oset)-
>rsm_name¬="rsm_q" then do;
                      new_args_ptr->retptr(kk)=mgr$cq(r1ptr(
kk));
```

```
                end;
                else new_args_ptr->retptr(kk)=addrel(r1ptr
(kk),r1ptr(kk)->oset)->quart_location;
            end;
            go to no_call;
        end;
        do kk=1 to limit;
            if q1ptr(kk)¬=null then do;
                new_args_ptr->q1ptr(kk)=q1ptr(kk);
            end;
            if q2ptr(kk)¬=null then do;
                new_args_ptr->q2ptr(kk)=q2ptr(kk);
            end;
            if r1ptr(kk)¬=null then do;
                new_args_ptr->q1ptr(kk)=pointer(addrel(r1p
tr(kk),r1ptr(kk)->oset)->quart_location,old_area);
            end;
            if r2ptr(kk)¬=null then do;
                if addrel(r2ptr(kk),r2ptr(kk)->oset)->rsm_
name¬="rsm_q" then
                    new_args_ptr->q2ptr(kk)=mark$q(r2ptr(k
k));
                else new_args_ptr->q2ptr(kk)=pointer(addre
l(r2ptr(kk),r2ptr(kk)->oset)->quart_location,old_area);
            end;
            new_args_ptr->r1ptr(kk),new_args_ptr->r2ptr(kk
),new_args_ptr->retptr(kk)=null;
            new_args_ptr->eptr(kk)=eptr(kk);
        end;
        call rsm_wq(new_args_ptr);
no_call: do kk=1 to limit;
            quart_ptr=new_args_ptr->retptr(kk);
            area_size=divide(quart_ptr->length*quart_ptr->
order+1050,1024,17,0)*1024;
            call area_(area_size,retptr(kk));
            allocate header in (new_area);
            rsm_name="rsm_q";
            if op="N" then go to skipit;
            alloc_length=quart_ptr->length;
            alloc_order=quart_ptr->order;
            allocate quart set (use) in (new_area);
            addrel(retptr(kk),retptr(kk)->oset)->quart_loc
ation=offset(use,new_area);
            do jj=1 to alloc_order;
                use->cent(jj)=quart_ptr->cent(jj);
                use->pent(jj)=quart_ptr->pent(jj);
                do mm=1 to alloc_length;
                    use->tent(mm,jj)=quart_ptr->tent(mm,jj
);
                end;
            end;
            if op¬="R" then free quart_ptr->quart;
        end;
        free new_args_ptr->args;
skipit:  end;
```

PRELIMINARY VERSION

```
tree_restructure: procedure (dsm_args_ptr);

  dcl data_seg area (area_size) based (seg_ptr);

  dcl 1 quart based (quart_ptr) ,
        2 id fixed binary (35) initial(0),
        2 length fixed bin (35),
        2 order fixed bin (35) initial(2),
        2 cent (2)  char (32) ,
        2 pent (2)  ptr,
        2 tent (alloc_length refer (length),2) fixed bin (35
);

  dcl 1 dsm_args based (dsm_args_ptr) ,
        2 upper_bound fixed binary (35) initial(1),
        2 op char (1) initial ("N"),
        2 di char (1) initial (" "),
        2 e char (1) initial (" "),
        2 d_or_q_ptr  ptr,
        2 ddptr  ptr,
        2 return_ptr  ptr;

  dcl 1 type based (type_ptr) ,
          2 dsm_name char(16),
        2 data_type char (4),
        2 num_free_cells fixed bin (35),
        2 num_entries fixed bin (35),
        2 max_length fixed bin (35),
        2 first_entry offset (data_seg),
        2 successor_chain_head offset (data_seg);

  dcl 1 item based (iptr) ,
        2 lptr offset (data_seg),
        2 rptr offset (data_seg),
        2 succ offset (data_seg),
        2 flag fixed binary (35),
        2 refno fixed bin (35),
        2 string_structure,
          3 string_length fixed binary (35),
          3 string character (max_length);

  dcl 1 area_fake based (fake_ptr),
        2 first_off  bit (18) unaligned,
        2 tprewq_fake  bit (18) unaligned,
        2 curr_len  bit (18) unaligned,
        2 qeeudm_fake  bit (18) unaligned,
        2 next_offset  bit (18) unaligned;
```

```
          declare (new_ptr,xfer,new_hdr) pointer;
          declare (next,temp_offset) offset (data_seg);
          declare (new_item_offset,last_new_item_offset) off
set (new_dd_area);
          declare (left_save,answer) (35) offset (data_seg);

          declare (off_hdr,alloc_length,qsize,qplace,level,c
um,nn,expo) fixed bin(17);
          declare (stem,number) (35) fixed bin(35);
          declare new_dd_area area (area_size) based (new_dd
_ptr);
          declare back (35) label (top_of_tree,coil_recur,fl
oor_recur);
          declare new_q_area area (area_size) based (new_q_p
tr);
          declare (wdir,input_place) char(168);
          declare err_code fixed bin(17);

          allocate dsm_args set(new_ptr);
          call hcs_$fs_search_get_wdir(addr(wdir),jjj);
          call hcs_$make_seg(wdir,data_type||"_GS_new",data_
type||"_GS_new",1011b,new_dd_ptr,err_code);
          if err_code>0 then do;
              call ioa_("Error has occured in allocating new
 segment for data type ~4a.File error code= ~6o.",
                   data_type,err_code);
              call ioa_("When you wish to continue, type any
thing and hit return.");
              call ios_$read_ptr(addr(input_place),1,jjj);
          end;
          call hcs_$make_seg(wdir,data_type||"_conversion_qu
art",data_type||"_conversion_quart",1011b,
              new_q_ptr,err_code);
          if err_code>0 then do;
              call ioa_("Error has occured in allocating new
 segment for conversion quart.");
              call ioa_("Data type is ~4a. File error code i
s ~4o.",data_type,err_code);
              call ioa_("When you wish to continue, type any
thing and hit return.");
              call ios_$read_ptr(addr(input_place),1,jjj);
          end;
          type_ptr=addrel(ddptr,ddptr->first_off);
          new_hdr=addrel(new_dd_ptr,new_dd_ptr->first_off);
          alloc_length=num_entries;
          qsize=2*alloc_length+1024+25;
          qsize=divide(qsize,1024,17,0)*1024;
          call area_(qsize,new_q_ptr);
          allocate quart_set(quart_ptr) in (new_q_area);
          pent(1),pent(2)=null;
          cent(1)=data_type;
          cent(2)=data_type||"_GS_new";
          return_ptr=quart_ptr;
```

```
          call hcs_$make_ptr("",dsm_name,dsm_name,xfer,err_c
ode);
          if err_code>0 then do;
              call ioa_("Illegal call to tree restructure -D
SM for old version of data type does not exist.");
              call GS_error$cr(99,"T",dsm_name,null);
          end;
          call cu_$ptr_call(xfer,new_ptr);



new_tree: number(1)=alloc_length;
          next=successor_chain_head;
          stem(1)=10000000000000000000000000000000b;
          back(1)=top_of_tree;
          qplace=1;
          level=1;
          go to optim;


optim: /*procedure (number,stem) returns answer*/
          if number(level)=0 then do;
              answer(level)=nullo;
              go to back(level);
          end;

          if number(level)=1 then do;
              allocate item in (new_dd_area) set (new_item_o
ffset);
              new_item_offset->flag=1;
              last_new_item_offset->succ=new_item_offset;
              last_new_item_offset=new_item_offset;
              answer(level)=next;
              new_item_offset->ptr,new_item_offset->rptr=nu
llo;
              new_item_offset->refno=stem(level);
              tent(qplace,1)=next->refno;
              tent(qplace,2)=new_item_offset->refno;
              qplace=qplace+1;
              if qplace=1 then new_hdr->successor_chain_head
=new_item_offset;
              temp_offset=next->succ;
              next=temp_offset;
              go to back(level);
          end;


          level=level+1;
          number(level)=divide(number(level-1)+1,2,17,0);
          floater=divide(number(level-1)+1,2,17,0);
          if floater>float(number(level),17) then number(lev
el)=number(level)+1;
          expo=max_depth-level;
```

```
            cum=1;
            do nn=1 to expo;
                cum=cum*2;
            end;
            stem(level)=stem(level-1)-cum;
            back(level)=ceil_recur;
            go to optim;
ceil_recur: left_save(level)=answer(level);
            answer(level-1)=next;
            temp_offset=next->succ;
            next=temp_offset;
            number(level)=divide(number(level-1)+1,2,17,0);
            expo=max_length-level;
            cum=1;
            do nn=1 to expo;
                cum=cum*2;
            end;
            stem(level)=stem(level-1)+cum;
            back(level)=floor_recur;
            go to optim;
floor_recur: allocate item in (new_dd_area) set (new_item_of
fset);
            new_item_offset->flag=1;
            new_item_offset->lptr=left_save(level);
            new_item_offset->rptr=answer(level);
            new_item_offset->refno=stem(level-1);
            last_new_item_offset->succ=new_item_offset;
            last_new_item_offset=new_item_offset;
            tent(qplace,1)=answer(level-1)->refno;
            tent(qplace,2)=new_item_offset->refno;
            qplace=qplace+1;
            if qplace=1 then new_hdr->successor_chain_head=new
_item_offset;
            level=level-1;
            go to back(level);
top_of_tree: last_new_item_offset=null;
            new_hdr->first_entry=answer(1);
            new_hdr->max_length=max_length;
            new_hdr->num_entries=num_entries;
            new_hdr->num_free_cells=0;
            return;
            end;
```

APPENDIX G

THE ERROR HANDLER

The error handler, GS error, is called via one of six entries, each of the basic form:

```
call GS_error$entry (error_number, one_character_error_
                     code, arg1, arg2)
```

The error number is printed out and also refers to the line in the file of error messages (GS_err_messages) which is to be printed out (e.g., error 51 prints out the 51st line). The one character code is the internal opcode to indicate to the user what operation was in progress. arg1 and arg2 depend upon the entry:

p, pr: arg1, arg2 are pointers
c, cr: arg1 is character, arg2 is a pointer
cc, ccr: arg1, arg2 are character

Null pointers are not printed out; i.e., their absence indicates they were null.

If one of the entries ending in "r" is used, the user is given the option to return from GS_error. The action taken then depends solely upon the invoker of GS_error. In any case, the user is given the options of "quit" and transfer to a procedure. This is done via the request:

"Type procedure name, or quit...." or with return allowed

"Type procedure name, return, or quit...."

The procedures most frequently invoked will be db and gsdb (see Appendix H). Two typical error cases are given:

```
call  GS_error$p(121, "U", lptr, rptr)
```

gives

```
GS:  Error 121 Internal Opcode U
     Pointer 1:  161|232   Pointer 2:  161|420
     Input quarts (pointers 1 & 2) disagree on choice
     of RSM.  Type procedure name or quit....
```

```
call  GS_error$ccr(221, "G", ent, cent(jj))
```

gives

```
GS:  Error 221 Internal Opcode G
     Name 1:  address      Name 2:  address
     Discrepancy between ent (name 1) and cent (name 2).
     Return yields null result.
     Type procedure name, return, or quit....
```

One final note: typing "quit" causes signalling of the condition "GOLD_STAR", so if the user should desire further processing of error conditions, he need only claim this condition. When a procedure name is typed, and that procedure returned from, GS_error reissues the request for a procedure name or quit (or return, if called for).

APPENDIX H

GOLD STAR DEBUG (gsdb)

The utility program gsdb is used to provide a tool with which to debug programs written for GOLD STAR. It may be invoked as a command, or as a subroutine, and will dump to the console four types of items: equivalence vectors, quarts, RSM argument structures, and DSM argument structures. The latter two are produced by the manager for the RSM's and DSM's and are useful to examine when a bug occurs within an RSM or DSM.

From command level, the user types "gsdb". The program responds with "ASK!". All requests are of the form:

X seg|offset

X can be e, q, r, d, or .. The first four correspond to equivalence vector, quart, RSM array, and DSM array; period indicates quit (see Appendix E for these structures).

When invoked as a subroutine, it is done on a per request basis; i.e., it is called gsdb$X (ptr) where X is as above.

The quit button can be hit to interrupt unwanted output; the command "pi" will resume gsdb at the request point.

APPENDIX I

SOME SIMPLE QUART EXAMPLES

This appendix is intended to give the reader a few examples of each of the basic operations as performed on quarts.

(1)

a	b	c
1	2	3
4	5	6
7	8	9

union

a	b	c
7	8	9
10	11	12
13	14	15

=

a	b	c
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15

(2)

a	b	c
4	5	6

union

a	b	c
1	2	3

=

a	b	c
1	2	3
4	5	6

(3)

a	b	c
1	2	3
4	5	6

b	c	a
2	3	1
7	8	9
10	11	12

←note the sort

=

a	b	c
1	2	3
9	7	8
12	10	11

(See also sort examples to clarify -- the second quart is sorted before union is done to the order a-b-c.)

(4)

a	b	c
1	2	3
4	5	6
7	8	9

intersect

a	b	c
7	8	9
10	11	12
13	14	15

=

a	b	c
7	8	9

(5)

a	b	c
1	2	3
4	5	6

intersect

a	b	c
7	8	9
10	11	12

=

a	b	c

(6)

a	b	c
1	2	3
4	5	6
7	8	9

difference

a	b	c
10	11	12
13	14	15

=

a	b	c
1	2	3
4	5	6

(7)

a	b	c
1	2	3

difference

a	b	c
4	5	6

=

a	b	c
1	2	3

(8)

a	b	c
1	2	3
4	5	6

difference

a	b	c
1	2	3
4	5	6

=

(9)

a	b	c
1	2	3
1	2	3
4	5	6
4	5	6

cart_prod

d	e
7	8
9	10

=

a	b	c	d	e
1	2	3	7	8
1	2	3	9	10
4	5	6	7	8
4	5	6	9	10

(10)

d	e
7	8
9	10

cart_prod

a	b	c
1	2	3
4	5	6

=

d	e	a	b	c
7	8	1	2	3
7	8	4	5	6
9	10	1	2	3
9	10	4	5	6

	a	b	c		b	a	c		b	a	c
	1	2	3	sort	100	101	102		2	1	3
(11)	4	5	6	(with respect to)				=	5	4	6
	7	8	9						8	7	9

	a	b	c		c	b	a		a	b	c
	1	2	3		100	101	102		3	2	1
(12)	4	5	6	sort (w.r.t.)				=	6	5	4
	7	8	9						9	8	7

	a	b	c		a	b	c		a	b	c
(13)	1	2	3	successor	7	8	9	=	10	11	12
	4	5	6								

	a	b	c		a	b	c		a	b	c
(14)	7	8	9	successor	a	b	c		a	b	c
	10	11	12		3	1	7	=	4	5	6

					a	b	c		a	b	c
(15)				successor	a	b	c		a	b	c
					10	11	15	=			

					a	b	c		a	b	c
(16)				successor	a	b	c		a	b	c
					1	1	1	=	1	2	3

Note: The following operations will automatically sort the second argument with respect to the first in order to perform the operation (see example 3): union, intersect, difference, successor relation.

		a	b	c						
(17)		a	b	c						
	get_ent	1	2	3	=	(3)	a	b	c	(equivalence vector)

(18)

a	b	c
1	2	3

modify ent (3) x y z =

x	y	z
1	2	3

(19)

a	b	c
1	2	3

modify ent (3) x & & =

x	b	c
1	2	3

(20)

a	b	c
1	2	3

clear_ent =

a	b	c
1	2	3

where all PENT's = null

(21)

a	b	c
1	2	3
1	2	4
1	3	5

project (3) x " " z =

x	z
1	3
1	4
1	5

(22)

a	b	c
1	4	5

project (3) & "" "" =

a
1

(23)

project (3) x & " " =

x	b
1	2
1	3
1	4

a	b	c
1	1	3
5	5	6
9	10	8
14	15	11

a	x	y
1	2	1
4	5	4
9	7	10
10	10	15

composition using equivalence vectors

(24) null vector ptr = a b c x y

1 1 3 2 1

9 10 8 7 10

(25) (6) m m & p & & = m & p x y

1 3 1 2 1

5 6 4 5 4

(26) (6) m m & m p q = m & p q

1 3 2 1

(27) (6) & w & & & w = a w c x

1 1 3 2

(28) (6) m n & & p n = m n & x p

1 1 3 1 2

10 10 8 9 7

15 15 11 10 10

Other operations do not lend themselves to simple quart usage.

REFERENCES

1-1 Smith, Burton Jordan, SPLP: A Special Purpose List Processor, Unpublished S.M. Thesis, M.I.T. Department of Electrical Engineering, Cambridge, Massachusetts, June 1968.

1-2 Fano, Robert M., "The Computer Utility and the Community", IEEE Convention Record, Part 12, 1967.

3-1 Rubin, Jean E., Set Theory for the Mathematician, Holden-Day, San Francisco, 1967.

3-2 Webster's Third New International Dictionary, G. & C. Merriam Company, Springfield, Massachusetts, 1967.

4-1 Vonhaus, A. H. and Wills, R. D., "The Time-Shared Data Management Systems: A New Approach to Data Management", SDC Report Sp-2747, Santa Monica, February 13, 1967.

4-2 Goldstein, R. C., Data Base Design Considerations, M.I.T. Project MAC, MacAIMS Multics Memo B.1, Cambridge, Massachusetts, February 12, 1970.

B-1 Martin, W. A. and Ness, D. N., Optimizing Binary Trees Grown With a Sorting Algorithm, M.I.T. Alfred P. Sloan School of Management Working Paper 421-69, Cambridge, Massachusetts, 1969.

BIBLIOGRAPHY

1. The MULTICS Programmers' Manual, M.I.T. Project MAC, Cambridge, Massachusetts, 1970.

2. Organick, E. I., A Guide to MULTICS for Subsystem Writers, M.I.T. Project MAC, Cambridge, Massachusetts, 1967.

3. Hays, David G., Introduction to Computational Linguistics, Elsevier, New York, New York, 1967.

4. Freyberghouse, R. A. et al, MULTICS PL/1 Language Specification, G. E. Cambridge Information Systems Laboratory, Cambridge, Massachusetts, 1969.

5. Freyberghouse, R. A. et al, A User's Guide to the MULTICS PL/1 Implementation, G. E. Cambridge Information Systems Laboratory, Cambridge, Massachusetts, 1969.

6. The MULTICS Systems Programmer's Manual, M.I.T. Project MAC, Cambridge, Massachusetts, 1970.

7. The CTSS User's Guide, Section AH 3.09, Ken Thompson, M.I.T., Information Processing Center, Cambridge, Massachusetts, 1966.

8. Thompson, Ken, Regular Expression Search Algorithm,

CACM, Volume 11, No. 6, June 1968.

9. Daley, Robert and Dennis, Jack, Virtual Memory, Processes, and Sharing in MULTICS, CACM, Vol. 11, No. 5, May 1968.

10. Graham, Robert, Protection in an Information Processing Utility, CACM, Vol. 11, No. 5. May 1968.

11. Bensoussan, A, Clingen, C. T., and Daley, R. C., The Multics Virtual Memory, Project MAC Memo MO111.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
Generalized Organization of Large Data-Bases; A Set-Theoretic Approach to Relations

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
B.S. and M.S. Thesis, Dept. of Electrical Engineering, M.I.T.

5. AUTHOR(S) *(Last name, first name, initial)*
Fillat, Andrew I. and
Kraning, Leslie A.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1970	250	7

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
ONR, N00014-69-0276-0002	MAC TR-70 (THESIS)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT Problems inherent in representation and manipulation of large data-bases are discussed. Data management is considered as the manipulation of relationships among elements of a data-base. A detailed analogy introduces concepts embodied in a data management system. Set theory is used to describe a model for data-bases, and operations suitable for manipulation of relations are defined. The architecture chosen for an implementation of the model is illustrated, and a representation of data-bases is suggested. A particular implementation, the GOLD STAR system, is investigated and evaluated. The framework outlined is meant to provide an environment in which complex data handling problems can be solved with relative ease. GOLD STAR provides the user with tools sufficient for manipulation of arbitrarily complex data-bases; these provisions are presented in the form of an extremely simple interface.

14. KEY WORDS
data bank, data base, data structure, data management, relations

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification