ABSPECT

THE STRUCTURE OF INFORMATION
IN GENERALIZED DATA-BASE MANAGEMENT SYSTEMS

by

MATTHEW FOWLER

One area of high concern in understanding the design of Generalized Data-Base Management Systems is that of understanding the nature of computerized information per se. This thesis is a theoretical investigation of that area. It presents a model of information that attempts to extend and consolidate the Entity Set Model of Senko et al., on which it is based. The method used is the translation of human information structuring concepts into computer representable terms.

Thesis Supervisor: Professor Stuart E. Madnick.
ACKNOWLEDGEMENTS

This thesis is the outgrowth of work started in the Sloan School Information Systems Group under Professor Madnick's guidance, which was in part supported by a UROP grant. I am grateful to Professor Madnick for supervising the thesis and bearing with all the administrative headaches it produced.

I am also indebted to Professor Louis Kampf, Professor J.H. Saltzer and John F. Callahan, without whose various kinds of support the thesis would not have been completed.

Any sentence expressing my thanks to Betsy, Don, Chris, and Susan would be insufficient.
### TABLE OF CONTENTS

Title Page ........................................... 1  
Abstract ............................................. 2  
Acknowledgements ...................................... 3  
Table of Contents .................................... 4  

1. INTRODUCTION ..................................... 5  

2. THE ENTITY LEVEL .................................. 11  
   2.1 Basic Ideas .................................. 11  
   2.2 Applied to the GDBMS ........................... 16  
   2.3 Descriptions ................................ 19  

3. THE SET LEVEL ..................................... 22  
   3.1 Sets and Information ............................ 22  
   3.2 The Communication Interface .................... 25  
   3.3 Sets .......................................... 28  
   3.4 SDL Defined in SDL ............................. 35  
   3.5 SDL in CIL .................................. 44  

4. NAMES .............................................. 46  
   4.1 Translation .................................. 46  
   4.2 Handling Names ................................ 47  
   4.3 Numbers ...................................... 57  

5. CIL REVISITED .................................... 61  
   5.1 Using CIL .................................... 61  
   5.2 Making Sense of CIL ............................ 67  
   5.3 Evaluating CIL ................................ 77  
   5.4 CIL in Perspective ............................. 80  

6. PROCEDURES ....................................... 84  
   6.1 Procedures and Virtual Information .......... 84  
   6.2 Procedures as Knowledge ....................... 88  

7. CONCLUSION ........................................ 94  
   7.1 Assumptions .................................. 94  
   7.2 Other Theoretical Work ......................... 95  
   7.3 The Real World ................................ 100  

Appendix 1: Memory Interface ......................... 103  
Appendix 2: BNF Definition of CIL ..................... 106  
Appendix 3: BNF Definition of SDL ..................... 108  

BIBLIOGRAPHY ......................................... 109
Generalized Data-Base Management Systems (GDBMSs) are currently the subject of much attention, commercially and in the research literature. Attempting a survey of the different ways of approaching the field and its results is beyond the scope of this thesis; introductions to the field from different view points are given in Folinus [Folinus, 73, Chaps. 2 and 4], Engles [Engles, 72], Senko et al [Senko et al., 73, Pt. 1], the Diebold Report [Diebold, 69] and Computerworld [Computerworld, 74]. The particular way we will approach GDBMSs is the subject of this chapter.

The term GDBMS is as much a slogan as a name for a class of existing systems. This is emphasized by the fact that we could have chosen almost any name-combination implied by Figure 1-1 and we would still be talking about approximately the same object. In order to put the motivation and scope of this thesis into perspective, we need to consider what sort of system the slogan aspect of these names attempts to describe.

One approach is to chart the evolution of the relationship between programs, systems and data. The dimension of this evolution is responsibility, the trend being to give the
system more and more responsibility, and so lessen the responsibility of the individual program. The first big step was taken by the file system, which assumed responsibility for the allocation, maintenance and accessing of secondary storage files. The file system essentially provided the program with an "auxiliary" memory independent of the actual storage media the system used, and delayed the binding of the symbolic file used by the program to the physical file until execution time. The responsibility for structuring and interpreting the file's data still rested with the program (or programmer). The evolutionary step attempted by the GDBMS is to assume this last responsibility. This is the goal of "data independence" (or "data structure independence" [Senko, 73, p.43] or "representation independence"). The program can then use a virtual file structure which will be independent of the actual structure of any physical file. The structuring and interpretation of the data files, and the
binding between the virtual and the physical structures at execution time, become the responsibility of the GDBMS. (For an elaboration of this view, see [Uhrowczik, 73] and [DBTG, 71, Section 1].)

This interpretation of the GDBMS slogan is mainly derived from a commercial orientation, and its motivation is to increase the efficiency of the programmer in relatively conventional applications (i.e. see [Computerworld, 74]). This is a limited view of the importance of the GDBMS, and we will not be directly concerned with it in this thesis.

A more fundamental way of proceeding is to consider the new relationship between the computer and the human intellect that the system implied by the GDBMS slogan promises. This relationship has undergone an evolution parallel to that of programs and data. The first step was made by the operating system of which the file system was a part, and time-sharing, which brought computing power within the range of the non-professional programmer because of the simplicity and responsiveness they afforded. Although this step was concurrent with and dependent on the file and operating systems, it was mainly motivated and guided by the conception of a "computer utility". This development is of course symbolized by MULTICS. [Corbato and Vyssotsky, 65]

Similarly, the promise of the GDRMS is more than merely data independence for the programmer. There is also
the promise of the "information utility", as it has been called. As the computer utility allowed the non-professional programmer to realistically conceive of the computer as an extension of her thinking power, so the information utility promises to enable the non-programmer to conceive of the computerized information system as an extension of her memory and senses. And this next step in the evolution of the relationship between human and computer should be guided by the explicit goal of the information utility.

It is not our job to discuss the nature of an information utility. The following quote gives a hint of the sort of system we have in mind:

"...We believe ultimately there will be very large systems serving tens of thousands of users. The users will be indifferent and largely oblivious to the fact that they are using a computer system. They probably won't even call their terminals "terminals"; names such as "fact-finder", "stockinfo", and "smartmart" are more likely. In particular, these people will not be programmers, mathematicians, engineers, scientists, or accountants. ... What we today might expect to be a command language will be the jargon of their particular environment, and they will be largely innocent about how the data they reference was acquired or is organized. [Dennis and Smith, 71, p. 11]

For a further discussion of information utilities see [Sprague, 69].

The realization of an information utility will be dependent on many facts. Hardware advances such as increased processing speeds and memory sizes and decreased hardware
costs are certain to take place. Software advances are much less a matter of course, and will in many cases depend on theoretical advances: new programming strategies for handling increases in software complexity; an understanding of file characteristics for automatic choice of file organisation; the construction of computer networks. It is probably no coincidence that these are all high-activity research areas today. This thesis is concerned with understanding the structure of information - another area of current interest, which will have a direct bearing on GDBMSs and hence on the realization of the information utility.

The importance of this topic is shown by the following quotes:

"...A global discussion of information is essential to an overall understanding of work in the information systems area." [Senko et al., 73, p. 46]

"A generalized data management system is a memory whose user view is designed to insure user convenience by representing data in its natural form. [Rothnie, 72, p. 16]

The major work in this area has been by Engles [Engles, 72], Codd [Codd, 70 and 71], and especially Senko et al [Senko et al., 73, Pt. 2]

Our basic approach is this. The natural view of a GDBMS is an obvious prerequisite for the acceptance of a GDBMS by the non-programmer. This natural view is an attempt
to make the GDBMS an effortless extension of human memory, which means that the structure of information in a GDBMS should replicate as nearly as possible human information structure. If this is possible, then in some respects the human brain is a GDBMS, which is the view we will take here. Our goal therefore is to understand the way we as human GDBMSs structure information, and then to map some of the human GDBMS concepts into computer representable concepts. A corollary of this is that these concepts and names for them should already be in common usage. Hopefully then, one result of this work will be to show that most of the confusing and unnatural terminology of the literature can be replaced by more natural everyday terms.

The tone of the thesis is somewhat informal and tutorial, for two reasons: I would like it to be readily comprehensible to people who are not information systems experts; and also, because the ideas in the area have been so obscured by the terminology barrier, the simplicity and naturalness of the ideas can only be demonstrated by starting from scratch at a very simple level. Our tutorial tone has two consequences. There are many examples and diagrams. I do not apologize for this - I see no other way to get across a feel for the proposed conceptual structure. Also, to keep the main development as clear as possible, the discussion of this paper's relation to other work in the area is left until Chapter 7.
II
THE ENTITY LEVEL

In this chapter, we want to introduce the basic ideas in our model of information, and discuss some primitive operations to support them.

2.1 Basic Ideas.

We will start with a simple example; I will describe a chair to you. It's a brown, wooden, hardback chair, which costs $25 and was made by the Sor Mill Corp. This is the sort of information we will be dealing with and trying to store and manipulate with a GDRMS.

What we are talking about here is a physical object, about which I have some information. In the paragraph above, I have tried to communicate this information to you. There are three distinct realms here. First, there is the object of discussion, in our case the chair, which with all due respect to the philosophers, exists independent of any knowledge we have of it. Then there is the information I have about the chair, which is stored in my brain. And lastly, there is the communication of this information to you. These realms are shown in Figure 2-1.

If we want to store some information in a GDRMS, we
will be directly concerned with realms (2) and (3), for we have to communicate with the GDBMS (3) to get at the stored information (2). We will be concerned with realm (1) only indirectly, as a realm to have some information about. For the rest of this chapter and the next, we will be concerned with the structure of realm (2). Our objective is to find the way we structure information as humans, and then map that123(218,532),(870,606)
reality (our realm (1)) is divided into separate entities. 

Another view from Engles:

"An entity is a person, place or thing. The thing may be real or abstract and this includes such things as events, classes, and relationships. As a matter of fact, anything can be an entity because an entity is simply that which we record facts about. [Engles, 72, p.13]

This last sentence points our way; the important thing for us is not the entities, but the facts we have about them. Here are the facts I gave about the chair:

<table>
<thead>
<tr>
<th>It is a member of</th>
<th>the set of chairs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Its style is</td>
<td>hardback.</td>
</tr>
<tr>
<td>Its color is</td>
<td>brown.</td>
</tr>
<tr>
<td>It is made from</td>
<td>wood.</td>
</tr>
<tr>
<td>It cost</td>
<td>$25.</td>
</tr>
<tr>
<td>It was made by</td>
<td>the Sore Mill Corp.</td>
</tr>
</tbody>
</table>

We have written these facts in this suggestive manner to illustrate our view of what a fact is. We restrict the word fact to mean a relationship between two entities; in the above facts, the relationships are in the middle column and the entities are in the outside columns. Other facts:

(Entity Relationship Entity)

| Mary is the sister of Martha. |
| Tiny's weight is 350 pounds. |

Notice that there is a direction implicit in the above facts. The relationship comes is the middle and points from the
first entity to the second, as shown in the Figure below.

![Figure 2-2.](image)

The ordering here is usually obvious to us, but it is important to get it right – it would be stupid to have a fact "The Sore Mill Corp. was made by this chair." So a fact is an ordered triple which we shall write as

Entity 1 / Relationship / Entity 2.

I will also sometimes refer to the Entity 2 here as the value of the relationship on Entity 1.

A relationship is our analog of a mathematical function. This is shown by Feldman and Rovner [Feldman and Rovner, 69] 's view of a fact as

Attribute (Object) = Value.

The form of this is obviously similar to \( f(x) = y \), and would correspond to

Relationship (Entity 1) = Entity 2 [Value].

Hence a relationship is very similar to a mathematical function. In normal speech it can usually be recognized by the keyword 'of', which is used in conjunction with it. "The color of the chair is red" indicates that "color" is a
relationship, "chair" is the object being described and the value is "red". The similarity of a relationship to a mathematical function is indicated by the fact that we read \( f(x) = y \) in the same way "\( f \) of \( x \) is \( y \)". The only difference between a relationship and a mathematical function is that the relationship maps from one element to many elements in general.

This is actually quite a restricted view of a fact. For instance, the information "The distance between the earth and the sun is 93 million miles", won't fit immediately into this form (though it can be put into an unnatural triple form). The same holds for "Jane's parents are Leslie and Bobby Jones".

Codd [Codd, 70] discusses a model information structure that can handle the above examples; on the other hand, Senko et al [Senko, 73] have essentially the same restriction. We have chosen to follow Senko partly to keep things simple and homogeneous conceptually, and partly with a view to implementation, as the restricted form obviously requires a lot less system overhead.

The last basic notion we have to introduce is that of a description. Our first example was really a description of an entity, the chair. Thus in general, a description of an entity is the collected set of facts that all refer to the entity in the first place; together the facts describe the entity.
2.2 Applied to the GDBMS

The first thing we have to do is get a way of representing entities, for the GDBMS is an information realm, so we can't have the actual entities in it! We shall map entities into Entity IDs, by which the GDBMS can identify and refer to entities. The form of an ID doesn't matter, because it will only be used internally by the GDBMS. The important thing is that for each entity we are concerned with, there is a corresponding unique ID in the GDBMS.

Another point is that relationships are a special type of entity, as Engles implies in the quote above. This means that internally a fact will appear as

Entity ID / Entity (Relationship) ID / Entity ID.

Looking at a relationship in this way gives a nice homogeneity: everything internally can be expressed in entity IDs.

We can now specify the lowest level interface we'll be concerned with. What we want below this interface is a "super-memory" which will support the ideas of the last section.

Logical Information Structure

---I-n-t-e-r-f-a-c-e---

Super-memory

Figure 2-3
The memory will be concerned with storing our information, while the levels above will be concerned with manipulating and communicating it.

Obviously the core of the memory's function is to store and retrieve facts for us. Then we can think of the memory's storage space as a long list of facts:

```
ID11 / ID12 / ID13
ID21 / ID22 / ID23
ID31 / ID32 / ID33
```

Figure 2-4

We can operate on the memory in one of two fundamental modes; we can **ADD** things to the list, or **DELETE** them from the list. Most GDBMSs have another common mode — **UPDATE**ing the list — which can be derived from the other two by first deleting and then adding. There is also the non-operational **QUERY** mode, in which we can ask questions about the state of the list, such as if a particular fact exists, if there is a relationship between two entities etc. Another way of looking at the distinction between operational and non-operational modes is in terms of information flow. In the operational modes information is going into the memory, whereas in QUERY mode we
are getting information out. In the **ADD** mode we can only specify complete facts to be added; it wouldn't do to try and add incomplete facts. In the other modes, we need considerably more power. For instance, we might want to find all the entity IDs which satisfy

\[ ? / \text{Entity ID} / \text{Entity ID}. \]

Or we might want to delete or retrieve the whole description of an entity with a single command. These examples illustrate the general principle that QUERY and DELETE requests can be given specifying only one or only two entity IDs as well as three.

Another thing we would like the super-memory to handle for us is the assignment and maintenance of the **IDs**. This is not a central part of the super-memory's function, but there is a good reason for assigning the job to it. The information structure to be set up above the interface does not depend at all on the actual content of the IDs, whereas the super-memory may be able to make good use of that content. For instance, an entity ID could actually be the disk address of the entity's description, or an integer that will hash into the entity's entry in a directory, or it could contain other information. So we will delegate this job to the super-memory in what follows and not concern ourselves with it.

The last thing we would like the super-memory to do for us is to check for redundancy. That is, we would like the
memory to only ADD a fact if the fact doesn't exist in storage already; it doesn't make sense to add the same thing again if it does.

This ends the general outline of the interface with the super-memory. A more detailed definition of the interface is given in Appendix 1. Clearly this super-memory is far more important in real life GDBMSs than its status in this paper would imply, and requires the most code and ingenuity to construct efficiently.

2.3 Descriptions

In the last section, we actually used only two of the ideas introduced in section 2.1 - facts and entities (and their IDs, of course). In this section, we want to make use of the idea of a description.

The trouble is that facts don't exist in a vacuum as the linear collection of Figure 2-4 would imply. Facts describe entities - all the facts about the chair describe (give information about) the chair in some way. Hence from now on, all our facts will be collected together under the ID of the entity they describe, which is the entity represented by the ID in the first place in the fact. Our chair example would look like Figure 2-5. All the character strings here (CON_TAINING SET, CHAIRS, etc.) stand for IDs. The facts have been put inside the box and condensed in the following way:
A CHAIR ID

CONTAINING SET / CHAIR
COLOR / BROWN
STYLE / HARDBACK
MATERIAL / WOOD
COST / $25
MANUFACTURER / SORE MILL

Figure 2-5

the ID of the entity (A_CHAIR_ID) is put above the box description, and it is implicitly included in the first place of the facts contained in the description. (Quite often, the entity's ID will be unimportant and will be omitted). So the first fact represented above would be A_CHAIR_ID / CONTAINING SET / CHAIRS in full. Note that there is no ordering implied by the position of the facts in a description as far as we are concerned. For instance, putting the STYLE fact before the COLOR fact in Figure 2-5 would not have changed the description.

This gives us a more powerful way of viewing the super-memory. We can now conceptually consider it to be divided into descriptions which contain all the facts about an entity, as shown in Figure 2-6. We now have more than facts; we have facts about entities, which is the most basic human
method of organising and classifying the information we have about the world.

\begin{figure}[h]
  \centering
  \begin{tabular}{|c|c|c|}
    \hline
    ID1 & ID2 & IDn \\
    \hline
    R11 / V11 & R21 / V21 & Rn1 / Vn1 \\
    R12 / V12 & R22 / V22 & Rn2 / Vn2 \\
    \vdots & \vdots & \vdots \\
    \hline
  \end{tabular}
  \caption{Figure 2-6}
\end{figure}
3.1 Sets and Information

Dividing reality into distinct entities is the first step in conceptualizing our experience; the second step is classifying entities into sets, which is our business in this chapter. This is illustrated by a quote from John Holt's delightful book, *How Children Learn*:

"A perceptive psychologist, Dr. Herman Witkin, in a book called *Psychological Differentiation*, aptly described the world of the young baby as "undifferentiated". It can't be broken down in parts. But as a baby gets older, he begins to see the room as a collection of things that are separate. Each object in the room - chair, table, lamp - has its own existence. It can be thought about by itself. When a baby makes this step, he is said to make an idea or mental model of the world that is differentiated.

Before he can start naming things, say a chair, the baby must take one more mental step. Not only must he see, first, that this chair exists by and of itself, independent of the room, that it could be somewhere else in the room, or in another room, he must also see that this chair is like certain other objects in the room and in other rooms. He must see that this chair is more like that chair than either of them are like a lamp, or a table or a rug.... In short, he sees that the chair is one of the family or class of like things." [Holt, 67, p. 57]

This "family or class of like things" we shall call a "set"; a set for us will be far more powerful than the mathematical concept, as we shall see.
Why are sets important to us? They are important to us as people because we use sets to simplify and order the world. We say "He's a laborer," "She's a business executive", "It's a medium-scale computer." Laborers, business executives and medium-scale computers are sets, and their use in the above examples gives a fundamental generic description in one word of the particular entities they refer to. Think of all the information encapsulated in the word "computer", for example. It takes a few months just to learn what the word really means; but then that information can be condensed into one word!

This raises two interesting points. One concerns man-GDBMS symbiosis. Although the set names convey much information, it is mainly the inexact sort of information that we humans thrive on; a present-day computer would have little use for it even if we could present it to a computer somehow. What we want to store in a GDBMS is not the intangible information that makes an entity a member of a set, but the (exact) facts about the entity that make it unique in the set.

Which indicates the second point we want to make about sets: they imply questions about the members. When I said "medium-scale computer" above, I thought about the following questions: Who was it made by? When was it designed? Is the CPU microprogrammed? What is the CPU cycle time? How
much main memory does it have? And so on. The answers to this type of question about a set member are what we want to store as facts in the entity's description. This information combined with the generic set information should tell us all we need to know about an entity.

Hence the most important way we shall use sets is for holding the questions we can ask about the entities in the set, and so providing a logical template for the descriptions of member entities. Furthermore, we will require that an entity must be in at least one set before we store information about it. For what sense does it make to store random facts about something (nobody knows what)? The facts would be completely unstructured, and as Senko says: "The real purpose [of a data-base system] is to store and present valuable structured information...." [Senko, 73, p.45; my underlining]. This structure for us is mainly the way sets organise the information in the member entities.

Figure 3-1 illustrates our conceptual view of the information we can store. Note that one of the entities in the figure is in two sets. This is the way we naturally think about entities, especially people. We are members of many sets, such as organisations, groups, cliques, etc., which all at some time or another help other people to understand or "place" us. So in our GDRMS, we would also like to allow entities to belong in more than one set.
This section has presented the most potent ideas for structuring information that we shall use. To summarize:

To be able to store facts about an entity it must be in at least one set. The name of the set gives us a lot of information which is common to all members of the set. This information is incomplete however, and leads to questions about the members. The answers to these questions describe a particular member entity more specifically than does the information common to all members. The storage of these answers as facts is the particular forte of a GDBMS. Figure 3-2 gives a pictoral summary.

3.2 The Communication Interface

Our objective for the rest of this chapter will be to sharpen and extend the ideas of the last section. A major aid in this will be the "languages" introduced in this sec-
tion and the next. In section 2.2 and Appendix 1, we sketched the lowest level interface, to support the idea of a fact. These new languages can be considered as a higher level interface, which allows us a more structured view of GDBMS information. As with all interfaces, it serves a double purpose. Conceptually an interface supports ideas; in this case the ideas of sets and entity descriptions. Functionally the interface is a communications vehicle; these languages will communicate between user and GDBMS.

Before developing these languages, we must make quite plain our orientation to them. We are interested only in them as tools to illustrate concepts, and not at all in their
syntax. Thus our motivation for giving BNF specifications for them (in Appendices 2 and 3) is that a BNF definition is another dimension for indicating a concept's scope and importance. Only in Chapter 5 will we rely at all (and then only marginally), on the BNF definitions; for that reason, they are relegated to the Appendices. We should also note that they are pedagogical devices; no constructs to handle real world environments are considered.

The more fundamental language, which we will call the Communication Interface Language (CIL), is the subject of this section. We will demonstrate the CIL by storing and manipulating the description of a new entity, a country. To enter the new entity's description we say:

NEW COUNTRY
(NAME = ATLANTIS,
POPULATION = 2,
PRESIDENT = HARRY OCTOPUS,
GNP = $1
AREA = 6,000,000 SQ.M.)

In terms of the graphic idiom of Chapter 2, this command would create Figure 3-3.
Examples of other commands are:

1. Deleting facts:
   
   `DELETE (POPULATION = 2) FROM ATLANTIS`

2. Deleting a whole description:
   
   `DELETE ATLANTIS`

3. Deleting facts by specifying relationships:
   
   `DELETE POPULATION, PRESIDENT FROM ATLANTIS`

4. Updating facts:
   
   `UPDATE (GNP = $2) IN ATLANTIS`

5. Retrieving selected information:
   
   `GET POPULATION, GNP OF ATLANTIS`

6. Retrieving all information on an entity:
   
   `DESCRIBE ATLANTIS`

7. Seeing if a fact is true:
   
   `TRUE (POPULATION = 3) IN ATLANTIS`

Hopefully the semantics of these statements are intuitively obvious. A BNF definition of the syntax of CIL is given in Appendix 2, which shows that there are more constructs to CIL than have been exhibited here; these constructs are discussed in Chapter 5.

3.3 Sets

You may have wondered if sets can be handled with the CIL. Figure 3-1 implies a sharp hierarchial distinction between sets and "normal" entities. So, can sets fit into our
normal way of thinking about entities?

We will take the view that in some cases they can, and some they can't! How can they? If you remember the first definition of an entity we gave, it was "Something that has separate and distinct existence, and objective or conceptual reality." By this definition, any set is an entity, because it has distinct conceptual reality. Also, as shown later in the chapter, it is convenient to consider a set as merely an entity in terms of a GDBMS implementation.

But it is equally important in another case to distinguish between sets and entities that are not sets. This is when we consider the way sets organise the information in the member entities. Then the information on a set not only describes the set, but also acts as a template. Clearly, we will need some additional structure to handle this.

Most existing GDBMSs have a rigid distinction between entities and sets. They have a language, equivalent to our CIL, which is often called the Data Manipulation Language, (DML), that is aimed at the entity description level. But they also have a Data Definition Language (DDL), which is used to define sets (for instance, see [DBTG, 71]). The words "Definition" and "Manipulation" capture the essence of the basic distinction between the two. This is that information on sets is generally more stable than information on their members. So it is appropriate to define sets but to manipulate
entity descriptions.

Because it is somewhat easier to talk about sets in a special purpose language, we will introduce one here, which we shall call the Set Definition Language (SDL). (SDL is really a conceptual convenience: an SDL command is in fact equivalent to a simple NEW command in CIL, as we shall demonstrate later in the chapter). The BNF definition of the syntax is given in Appendix 3; here we will again use an example to illustrate the semantics. The set of countries would be defined as follows:

DEFINE COUNTRY,
    NAME IS CHARACTERSTRING SINGLE_REQ,
    POPULATION IS NUMBER SINGLE_REQ,
    PRESIDENT IS PERSON SINGLE,
    GNP IS DOLLARS SINGLE
    AREA SINGLE
$END

The word after DEFINE is the name of the set, in our case COUNTRY. The subsequent lines all define possible relationships - NAME, POPULATION etc. - between the set being defined and other sets, such as CHARACTERSTRING, NUMBER etc. The situation here is analogous to mathematical function definition, f:A→B, where f corresponds to our relationship. We would define this setup:
Define
f IS R,
$END

The word "IS" in the definitions is an indication of the most illuminating way of reading them:

Define [a new set] COUNTRY
[with a relationship]
NAME [defined on it, whose value]
IS [a] CHARACTER_STRING
e tc.

The words SINGLE_REQ, REQ AND SINGLE we will refer to collectively as "repetition specifications": they specify limitations on the numbers of facts with this relationship that may appear in the description of a member of the set. They are again somewhat similar to the mathematical concepts of one-to-one, one-to-many, etc., but for us they are restrictions rather than statements of fact. The full list is:

SINGLE_REQ One and only one fact of this sort must appear in a member's description.
SINGLE At most one fact of this sort may appear.
REQ Stands for REQUIRED; at least one fact must appear.
REP REPeatable; no restrictions.

So in any description of a COUNTRY, we would find its NAME and POPULATION given exactly once, while the PRESIDENT, GNP
and AREA might be given, but not twice. We will use SINGLE_REQ as the default repetition specification.

What does this definition buy us? If "us" is a user, we can see what sort of facts a country will have stored about it by looking at the definition. But if "us" is the GDBMS system, this definition has more implications. For an important function of the GDBMS will be to validate the information it stores, and the set definition provides most of the templating data needed to do the checking. In Figure 3-4 the super-memory would only be needed to check that an entity with the exact same description hadn't already been stored - again, at this logical level, it makes no sense to store the same entity twice. All the other validity checking would be based

![Diagram](image-url)

Figure 3-4
on the set definition. There are three rules or checks implied by the definition:

1. That the member description contains no facts whose relationship was not given in the set description.

2. For each fact, the value of the relationship is member of the range of the relationship.

3. The numbers of facts in a description with the same relationship must satisfy the restrictions imposed by the repetition specifications, as detailed above.

These rules extend naturally to entities in more than one set. Only the first rule is affected, and the permissible relationships are in the union of the set descriptions. So the following attempt to enter a COUNTRY is in error at the rules noted in the comments.

```
NEW COUNTRY (NAME = ARTICA,
           NAME = ANTARTICA, [Rule 3]
           PRESIDENT = PETER PENGUIN [Rule 2 - Peter
                                           Penguin is a Penguin]
           AV_TEMP = -10°, [Rule 1]
         ) [Rule 3 - no POPULATION given]
```

Another example of SDL:

```
DEFINE COURSE,

       NUMBER IS CHARACTER_STRING,
```
HOURS IS INTEGER SINGLE,
LECTURER IS PROF REQ,
STAFF IS PROF REP,
DEFINE SECTION,
  NUMER IS CHARACTER_STRING SINGLE,
  MEETING_TIME IS TIME REQ,
  NUMBER OF STUDENTS IS INTEGER,
$END,
REQ,
DEFINE STUDENT_INFO,
  STUDENT,
  SECTION,
  CUM_GRADE IS CHARACTER SINGLE,
$END
REP
$END

We're talking about an academic course here, the variety that lasts for one semester. Two new ideas are introduced by the example, both of which first occur in the line "DEFINE SECTION". What we are doing is defining another set in the middle of defining the first; the definition lasts until the first $END encountered. STUDENT_INFO is another example of this. This type of definition is equivalent to a separate definition of the set, with the set name being inserted in its place in the outer definition. In-line definitions are
useful mainly because they should make it easier to read and understand the logical interdependence of the set definitions, than if the three sets were defined separately. Conversely, they should also be easier to write this way. In some GDBMS schemes, the above definitions would establish a hierarchical relationship between the COURSE set and the SECTION or STUDENT_INFO sets, so that COURSE would become the "parent" or "owner" SECTION and STUDENT_INFO. In our scheme, there is no need for such a hierarchical relationship and none is implied by the definition of the one set inside the definition of another.

The other new issue is syntactical rather than conceptual. It is that it is often useful to have the relationship defined merely by the range set name, as in the STUDENT and SECTION lines in the STUDENT_INFO definition, rather than having to specify the same name for both. We could have said "STUDENT IS STUDENT" in those lines, but it would have been redundant; so this capability is merely an aid to concise definition than anything else. Note that, in view of what was said in the previous paragraph, SECTION and STUDENT are further examples of this capability. An example of a COURSE description is given in Figure 3-5.

3.4 SDL Defined in SDL

As SDL, or any other DDL, is a language for the descrip-
tion of information structures, and as SDL can be itself considered as an information structure, in this section we will define SDL in SDL. There are two reasons for wanting to do this. First, this definition should "pin down" the semantics of SDL more tightly than is possible with words. But also, it will lead to a homogenous implementation in that "system" data-bases (sets) can be handled in exactly the same way as normal sets.

The bases of SDL are entities, sets and relationships. We will define them as sets here and then discuss the definitions.
DEFINE ENTITY,
    CONTAINING_SET IS SET REQ,
$END
DEFINE SET,
    NAME IS CHARACTER_STRING,
    SUPERSET IS SET REP,
    SINGLE_REQ IS RELN REP,
    SINGLE IS RELN REP,
    REQ IS RELN REP,
    REP IS RELN REP,
$END
DEFINE RELN, [Relationship]
    NAME IS CHARACTER_STRING,
DEFINE DR PAIR, [Domain-range pair]
    DOMAIN IS SET,
    RANGE IS SET,
$END,
REQ,
$END

Before discussing these definitions, we must note that they are "skeletons", in the sense that we have only given the bare bones of what we need for supporting the logical structure and for completeness; obviously a working GDBMS would use more fields than this in the corresponding set definitions.

-37-
The definition of ENTITY merely formalises our notion that everything we can describe is an entity. This has been assumed before, and we will continue to do so; that is, there will be no need to define any set as a subset of the ENTITY set. The interesting thing about the definition is the CONTAINING_SET relationship; this is the bootstrap by which we can store interesting facts about an entity.

The SET is the heart of the structure. We are defining the set of sets here - how a set can be described - so that saying "set" is equivalent to saying "a member of the SET set". This may seem confusing, but normally it is quite natural: "medium-scale computer" is the name of a set, but when we talk about a member of that set we would refer to it as "a medium-scale computer". Similarly, there is the concept of the SET, which generically describes a set. This leads to Figure 3-6, which is an updated version of Figure 3-2. We now have three levels instead of two. A set must have a NAME, which is the word after DEFINE in SDL. Any number of SUPERSETs may be defined after the NAME, as in "DEFINE MANAGER IS EMPLOYEE": this definition will ensure that every manager is also an employee. The effects of the last four lines was discussed before, in section 3.

The definition of RELN (for relationship) is straightforward. The DRPAIR (domain-range pair) gives the DOMAIN and RANGE sets for the relationship. Notice we defined the
DRPAIR as REQ rather than SINGLE_REQ. This would appear to be contrary to the mathematical definition of a function, which can only be defined on one set. However, although relationships would be defined on only one set normally, we may be interested only in subsets of the set. For instance, all the family relationships (brother, mother, etc.) are defined on the set of sexually reproducing organisms, very few species of which we would want to store information on. So we would like the brother relationship, say, to be definable on the set of dogs. (Snoopy has brothers and sisters too!). Another relationship we need this capability for is the NAME relationship, which we have already applied to many sets.

This requirement for multiple ranges and domains for a relationship leads to the awkward form of the definition.
Here is one case where having facts as triples (Entity/Relationship/Value) is constricting; it would be a lot cleaner in this case to define a double value:

\[
\text{RELN} / \text{DR~PAIR} / (\text{DOMAIN}, \text{RANGE})
\]

As it is, the only way to avoid pairing up the wrong \text{DOMAIN} and \text{RANGE} is to put the correct pairs into separate entities.

To show how the \text{SET} and \text{RELN} definitions are used, we will show the effect of the following set definition:

\[
\text{DEFINE S IS SS}, \\
R1 \text{ IS S1,} \\
S2 \text{ REP,} \\
\$\text{END}
\]

\(S\) is the name of the set, so \text{DEFINE S} produces:

\[
S \\
\text{CS / SET} \\
\text{NAME / "S"}
\]

"IS SS" means that \(SS\) is a superset of \(S\):

\[
S \\
\text{CS / SET} \\
\text{NAME / "S"} \\
\text{SUPERSET / SS}
\]

\(R1\) is now contextually defined as a relationship, with \(S\) as its range and \(S1\) as its domain, which will lead to an entity for \(R1\):
R1 must also be recorded in S's description as SINGLE_REQ:

S

S2 is also a previously defined set. But just saying S2 as a line in the definition also implies that S2 is the name of a relationship, as we pointed out on page 34. So this line will produce:

The relationship S2 is REPeatable on S, which gives the final result of Figure 3-7.

This should make it apparent how entities are templated
by the sets they are in. The values of the relationships in the set description are the relationships which become candidates for entry into a member's description, subject to the constraints imposed by the repetition specifications, as shown in Figure 3-8. Notice that this scheme also works for sets as members of the SET set. In Figure 3-9, the CS / SET fact is allowed because the object is an entity; all other facts conform to the definition of SET given above.

```
CS / SET
NAME / "S"
SUPERSET / SS
SINGLE REQ / R1
REP / S2RELN
```

**Figure 3-7**

```
X
Set  CS / SET
     NAME / "X"
     SINGLE REQ / R
Entity CS / X
     R / V
```

**Figure 3-8**
This scheme of SDL allows a touch of elegance, for the SET set is valid both as a set and as an entity. That is, if we follow the implied algorithms for generating a description of a set from its SDL definition, we get Figure 3-9 from the definition of SET. Only one subsidiary relationship description, of REP, is given; the others are similar. This is clearly a valid set and also a valid member of the SET set. So the SET set validly templates itself, and is validly a member of itself. With this example, we have completed the definition of SDL in SDL.

This may have seemed like an exercise in self convolution. However, we hope that this section and the next will make the semantics and scope of these languages as clear as possible. This section also has an implication for GDBMS construction. We have shown that a set can be considered as

<table>
<thead>
<tr>
<th>SET</th>
<th>REP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS / SET</td>
<td>CS / RELN</td>
</tr>
<tr>
<td>NAME / &quot;SET&quot;</td>
<td>NAME / &quot;REP&quot;</td>
</tr>
<tr>
<td>SINGLE_REQ / NAME</td>
<td>DR_PAIR / DR</td>
</tr>
<tr>
<td>REP / SUPERSET</td>
<td></td>
</tr>
<tr>
<td>REP / SINGLE_REQ</td>
<td></td>
</tr>
<tr>
<td>REP / SINGLE</td>
<td></td>
</tr>
<tr>
<td>REP / REQ</td>
<td></td>
</tr>
<tr>
<td>REP / REP</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-9
merely an entity if convenient. In validity checking and entity (set) input, and possible other operations, it is very convenient to do so; there is no need for two sets of "unsymmetrical" algorithms for sets and entities. Furthermore, there should be no need for "system tables" other than normal sets at this logical level. - the logical information structuring level could operate in a "data-less" way because all its information would be entrusted to the super-memory.

Another area where being able to define a DDL in itself may be useful is in the interchange of information in GDBMS networks. Via a common DDL, a GDBMS could describe its information structure - its SRT definition, etc - to another GDBMS. This would allow the GDBMS sending files to another to arrange them in such a way as to ensure their validity in the receiving system.

3.5 SDL in CIL

The above remarks make it important that SDL can be mapped into CIL. In fact, when we were introducing SDL, we said it was equivalent to a NEW command in CIL, and the last section should have made the mapping fairly obvious. In this section we specify this mapping explicitly.

Let us take our S example again:

DEFINE S IS SS,
    R IS S1 SINGLE_REQ,
S2 REP, $END.
The equivalent CIL command is:

```
NEW SET (NAME = S,
    SUPERSET = SS,
    SINGLE_REQ = NEW RELN
        (NAME = R1,
            DRPAIR = NEW DRPAIR
                (DOMAIN = S,
                    RANGE = S1)
        )
    )

REP = NEW RELN
    (NAME = S,
        DRPAIR = NEW DRPAIR
            (DOMAIN = S,
                RANGE = S2)
    )
```

There is only one case where SDL is not equivalent to CIL. This is in the "system" data-bases - SET, RELN, etc. Clearly, CIL and SDL are dependent on these definitions if they are to have any meaning. So it would make no sense to try to define SET in CIL, because it would have to be defined before the CIL could work. This emphasises the difference between the languages. Although if SDL is used on a GDBMS if effects a change in the data, its "tone" is static; it describes the state of things more than actions. By contrast, CIL is purely effective - its purpose is to change or retrieve the information in the data-base.
4.1 Translation.

In Chapters II and III we have been concerned exclusively with realm (2), the structure of information in the GDBMS. In this chapter, we consider the interaction between realm (2) and realm(3), communication.

The obvious prerequisite for this discussion is the definition of a translation mechanism. For up till now the only "data items" we have considered have been entity IDs, which are only meaningful to a particular GDBMS. So a translation mechanism would map between some entity IDs and externally valid identifiers such as character strings. The basic structure is shown in Figure 4-1.

![Figure 4-1](image-url)
4.2 Handling Names.

If the translation mechanism were the whole story concerning communication, this chapter would have ended already. However, there appears to be a need for a capability for handling names and communication, which is largely independent of this translation mechanism.

The human analogy will illustrate this. If I write "red", you will translate this into a name entity. But if I said "red", you would still translate this into the same name entity. So, using either your visual or auditory translation mechanism, you would arrive at the same name "ID". Then you try to map this name ID into the entity that it names - in communication, we're mainly concerned with identifying entities via their names. In this case, you would normally map the name "red" into the color red. However, in some circumstances, you might decide that "red" was the (misspelt) name of a friend "Red". This process is illustrated in Figure 4-2.

The point we want to emphasize about this example is that there are two easily distinguishable mechanisms here. Furthermore, underlying the example is the basic idea of identifying an entity. The whole idea of communication is to transmit information about specific entities, which implies that effective communication requires the unambiguous identification of entities. This explains the importance of names - surely this is the meaning of "name". So for our purposes
(we are concerned with written communication), we can define a name to be a character string or set of character strings that, in a given context, are meant to identify a single entity.

We can now specify more precisely the translation function of section 1. The function's domain is the set of character strings, and the translation mechanism must map back and forth between internal character string IDs and their external representations (character strings, sound
waves etc.). Furthermore, this mapping is one to one and onto - the translation of a valid representation in one space (internal or external) will yield exactly one representation in the other space. So the translation mechanism is conceptually very simple.

Mapping between entities and their names is much more complex. For a start, the mapping is many to many. For instance, many people have the same name (in the Boston Telephone Directory, there are four pages of Smith's, 37 John Smith's, and 5 John J. Smith's); and most people have a human name and a social security number. On top of this, there is the problem of the different ways we handle names. The first problem can be handled quite easily within the structure we already have. To allow an entity to have many names we could use a combination of NAME and SYNONYM fields - as in

```
DEFINE SET,
  NAME IS CHARACTER_STRING SINGLE_REQ,
  SYNONYM IS CHARACTER_STRING REP,
  etc.
```

(The unique NAME allows the system to know which name to use in identifying the set to a user.) And to be able to get back to many entities from one name, we could

```
DEFINE CHARACTER_STRING
  ENTITY_NAMED IS ENTITY, $END
```

so that when the translation of an external character string
yields a CHARACTER_STRING ID, the GDBMS could easily find all the entities whose description the character string was used as a name in. (The ENTITY_NAMED field is a conceptual convenience: the implied capability could also be achieved by asking the interface of Chapter II for all IDs that fit ? / NAME / THE_STRING.)

The most basic way of handling names is rather simple: entities we want to identify have names, which is signified by a NAME relationship in the set's description. One example of this is the definition of SET given in Chapter III. Another is

```
DEFINE COLOR,
      NAME IS CHARACTER_STRING,

-END
```

Each color is conceptually a separate entity, and we are only interested in a color's name. Now although this is the way our thinking works, in a GDBMS this structure would be cumbersome if we were only interested in the entity's name. What we would prefer to do is to bypass the entity's description, and just store the name. So instead of saying

```
DEFINE AUTO,
      COLOR,

etc,
```

we would rather say
DEFINE AUTO,

COLOR IS CHARACTER_STRING,

etc.

If we aren't interested in COLOR as a set, the latter scheme is equivalent to the former and has the advantage of yielding the name of the color - which is what we will eventually need anyway - immediately.

Thus we have two schemes for naming: an explicit NAME relationship, whose range is a character string, in an entity's description; and the implicit naming of defining the range of a relationship to be a character string. We arrive at the need for a third scheme, because some entities have names that are not simple character strings. The obvious example of this is a person's name, which has three distinct components. Another example is an address - although addresses are not used in a NAME relationship, they are names by our definition because an address is a set of characters which designates a unique entity.

The difficulty then in handling names is not the supporting information structure - we already have this. The trouble is that we use names as identifiers in any of the above four ways without consciously indicating which, and that computer implementations require this usage to be clearly specified. As an example of our unconscious use of names, let's reconsider the way we used names in the last chapter.
Figure 4-3 reproduces part of Figure 3-9. Now what do most of these "names" here (SET, CS, NAME, etc.) represent? They represent entity IDs. In most of the cases above, the external names (which are the same as the representations above) can be retrieved by getting the value of the NAME relationship in the entity's description (i.e. "SET"), which will be a character string ID, and translating it into external form. "SET" of course is different; it represents the character string ID that will translate into the external string "SET". This convention (which we have used until now and which we will continue to use without mention) probably caused you no problems, because it mirrors the way we naturally use names as identifiers. But notice that you mapped the names into entity IDs even if you didn't realize it. This mapping, which is illustrated in Figure 4-4, is what we must make explicit.
We note that the mapping from an external character string to the internal entity ID is more complex than the inverse operation in general because the ENTITY_NAMED relationship for a character string may yield more than one entity ID. We will leave consideration of this process till the next chapter.

To bring these ideas together, we now give an example. We will use the following sets:
DEFINE PERSON,
    NAME IS DEFINE PERSON_NAME
    FIRST_NAME IS CHARACTER_STRING,
    INITIAL IS CHARACTER,
    SURNAME IS CHARACTER_STRING,
$END,
DEFINE ADDRESS,
    OCCUPANT IS PERSON,
    NUMBER,
    STREET IS CHARACTER_STRING,
DEFINE CITY,
    NAME IS CHARACTER_STRING,
    POPULATION IS NUMBER,
$END
ZIPCODE IS NUMBER,
$END
$END

In the ADDRESS set, we defined the range of the STREET relationship to be the set of CHARACTER_STRINGs, whereas CITY is a set in its own right. Clearly, this doesn't reflect our way of thinking about streets - a street is an entity too. This definition merely means we are interested in facts about cities but not about streets. The example entities from these sets that we will be concerned with are shown in Figure 4-5.
We want to see how the GDBMS will reply to the following commands:

1. GET (STREET OF ADDRESS) OF FRED A. STARE.
2. GET (CITY OF ADDRESS) OF FRED A. STARE.
3. GET OCCUPANT OF 11, MAIN ST., HICKSVILLE.
4. GET ADDRESS OF FRED A STARE.

We are not concerned at present with how the GDBMS translated "FRED A. STARE" and "11, MAIN ST., HICKSVILLE" into entity IDs - clearly it should come up with PID and AID; we are initially concerned with the IDs that result from the evaluation of the commands. In the first and second commands, we have used two relationships in sequence, which we haven't encountered before. A full explanation of CIL is given in the
next chapter; for now, we note that the ADDRESS of FRED is AID (from the fact PID / ADDRESS / AID), and the STREET of AID is "MAIN". See Figure 4-6. The other commands are evaluated similarly to yield the IDs "MAIN", HICKSVILLE, PID and AID.

"FRED A. STARE"  

Entity to Name Mapping  

ADDRESS OF  

(PID / ADDRESS/ AID)  

AID  

STREET OF  

(AID / STREET / "MAIN")  "MAIN"  

Figure 4-6.

The problem now is to give the user an answer. Clearly "MAIN" should be given straight to the translator because it is a character string ID. The HICKSVILLE ID will not have a translation; this is an example of our simplest case – the entity in question has a NAME. So in this case the GDBMS will retrieve the value of the NAME, "HICKSVILLE", and translate that. PID also has a NAME, but its value, PNID, is not a character string. The last value, AID, is even worse: it is not a character string and it doesn't have a NAME field!
These are examples of our "non-simple" names. They are names in the sense that, from IDs, we must construct some output that will identify the entity for the user. We can only hint at the solution here: what is needed is a procedure that will take the ID as input and produce the extended name output. For example, PID might be converted into "FRED STARE". The place of procedures in the GDBMS scheme will be considered in Chapter VI.

As a result of this discussion, we can imagine the GDBMS using the algorithm of Figure 4-7 to convert the ID results of a GET command into user readable form.

4.3 Numbers.

What we implied in the last section is that, for every character string known to the system, there is an internal/external entry in the translation data-base. However, this isn't generally true. For in the preceding discussion we were assuming that the character strings were words. But numbers have character string names too, and clearly it is impossible to store the names of all the numbers in the translation data-base.

We are therefore led to consider the difference between numbers and other entities we have been talking about. An immediate difference is that numbers are not the sort of entity we would want to store facts about: "The
numbers by themselves are relatively worthless." as a professor once declared. But in conjunction, they are worthwhile, as the quote implies. Clearly, numbers are much used as values in facts (i.e. DEMIAN / AGE / 2, BOLT / WIEGHT / 51) because of the way they are used to order objects and quantify relationships.

-58-
Numbers are also useful in connection with algorithms. For instance, if we see $f: \mathbb{R} \to \mathbb{R}$, we usually infer that $f$ is algorithmically computable. That is, given a representation of $x$ and **no other information**, there is an algorithm to compute a representation of $f(x)$. Now, this is only possible if $\mathbb{R}$ is a set of numbers, to put it loosely. If $\mathbb{R}$ is a set of people, there is essentially no relationship whose value we can find given just a person's name, which is why we need data-bases to store the answers. This is not to say that for arbitrary sets we cannot evaluate $f(x)$; it merely means that we have to have the value already stored, and do the evaluation by "table-lookup". The major difference between numbers and other sets that makes this data-less computation possible for numbers is that the meaning of a number is so close to its representation.

So we have distilled two main differences between numbers and other entities. First, numbers are only interesting as values of relationships; the numbers in themselves are not worth storing information about. Second, the essence of numbers and their names are very closely (algorithmically) related, which is not true in general. This said, the actual handling of numbers by a GDBMS is anticlimactically simple. While the translation module will need a data-base to translate between external character strings and character string IDs for other names, for numbers this is not necessary — all that is needed is a conversion program similar to the ones
in general usage today, which in our terms do convert between internal IDs and external character strings. Furthermore, as the numbers by themselves are "relatively worthless", there is no need to have entity descriptions for them as long as the GDBMS can "recognize" a number ID. This can be accomplished quite trivially by, for example, forming a number ID by concatenating a normal integer or real variable bit pattern with an INTEGER or REAL set identifier (as is done in the Burroughs B5000 series [Burroughs, 61]). As a result of this, we can also assume that a GDBMS will be able to do arithmetic operations on number IDs directly.
V
CIL REVISITED

In this chapter, we show the full power of CIL, as we promised in Chapter III. The examples of CIL we gave there illustrated the basic operations of the entity level; we now extend CIL's range of operation to the set level. We give more examples in the first section, and show the underlying GDRMS capabilities which support the CIL in the next two sections.

5.1 Using CIL.

In order to extend CIL to the set level, we will need to introduce new concepts into the language. The purpose of this section is to introduce the concepts informally and intuitively; more formal information about the constructs can be found in the BNF definition in Appendix 2. The examples in this section are based on the examples given in Section 3 of [Codd, 71]. The sets we shall use are slightly modified versions of his:

```
DEFINE SUPPLIER,
   NUMBER,
   NAME,
   LOC IS ADDRESS,
```
Codd's system does not place much importance on repetition specifications, so we will assume the default (SINGLE_REQ) for all the relationships above. The example operations are headed by the number of the operation in Codd's paper and his description of it in English.
3.1 "Find all the part numbers of parts being supplied."

GET NUMBER OF PART ( ).

"PART ( )" is an example of an entity specification — in this case a specification of all parts. Hence, "PART ( )" will be evaluated to the set of PART IDs, and the NUMBER relationship applied to this set will give the set of IDs that are part numbers of parts being supplied. If a number is the number of two or more parts, it will only appear once in the resulting set of number IDs.

3.2 "Find the part numbers, names and quantities on hand where quantity on hand is less than 25."

GET NUMBER OF SUPPLIER OF SUPPLY (QOH < 25)

3.6 a) "Find the supplier numbers of those suppliers who supply the part with part number 3."

GET NUMBER OF SUPPLIER OF SUPPLY (NUMBER OF PART = 3).

"NUMBER OF SUPPLIER" and "NUMBER OF PART" are examples of what we will call compound relationships, which we already encountered in Chapter IV. A compound relationship — X of Y — makes sense only if the domain of X is the range of Y; it is then a relationship between the domain of Y and the range of X. This situation is illustrated in Figure 5-1.

These last two examples hint at how the specification of an entity is to be viewed. Essentially, the lines inside the parentheses after the set name are predicates, despite
their similarity to facts when the "=" sign is used; each of the predicates must be true for an entity if it is to be included in the set of IDs specified by the expression. We will define this more precisely later; for now we note that the form of the specified predicates can be more complex than the form of simple facts that would be in an entity description.

Figure 5-1.

3.7 "Find the supplier numbers of those suppliers who have the same location as supplier Jones."

GET NAME OF SUPPLIER (LOC =
  LOC OF SUPPLIER (NAME = JONES) )

This example shows a more complex predicate. "SUPPLIER :
(LOC = LOC OF SUPPLIER (NAME = JONES))" denotes all the suppliers whose description satisfies the predicate: his
location is the same as the location of a supplier named Jones.

3.10 "Find the names and locations of all suppliers each of whom supplies all projects."

GET NAME, LOC OF SUPPLIER ( (RELN (S) PROJECT OF SUPPLY (SUPPLIER=S)) SETEQ PROJECT ( ) )

SETEQ is another relational operator, which allows the predicate to be true only if the two sets it is being applied to have the same members - which would be all project IDs here. Other relational operators at the set level are SUBSET and SUPERSET, for the obvious reasons.

To isolate the meaning of (RELN (S) ... = S) ), we can set up a correspondence between the command in English and in CIL. "Find the names and locations of all suppliers" clearly maps into "GET NAME, LOC OF SUPPLIER ( )", leaving the predicate to be filled in. "All projects" becomes "PROJECT ( )"; so that "each of whom supplies" is what is left. Rephrased into set languages, this is "the set of projects that the supplier supplies". Now, this phrase implies a mapping from the SUPPLIER set to the PROJECT set. But the only major set we have introduced that has relationships to another is the SUPPLY set, which has relationships to the other three sets, (SUPPLIER, PART, PROJECT). The
relationships defined on SUPPLIER are NUMBER, NAME and LOC, which all appear to be irrelevant here. This is why, in order to get the desired mapping, we have to define it ourselves - which is precisely what the relationships definition construct does. Any similarity between this construct and function definition in LISP etc. is purely intentional, and only serves to emphasize the operational similarity between relationships and functions. "S" here is a dummy "variable", which can represent the SUPPLIER IDs, as the relationship is evaluated for them. So for each SUPPLIER, "(RELN (S) PROJECT OF SUPPLY (SUPPLIER = S) )" will be the set of PROJECTS that appear in the same SUPPLY description as the SUPPLIER - in other words, all the projects that the supplier supplies.

3.11 "For each project obtain as a triple the project number, project name and supplier location for all suppliers who supply that project".

GET NUMBER, NAME, (RELN (P) LOC OF SUPPLIER OF SUPPLY (PROJECT = P) ) OF PROJECT ( )

The relationship definition here gets around exactly the same problem as before - that there is no direct mapping from PROJECT to SUPPLIER (and hence to the suppliers location).
5.2 Making Sense of CIL

Hopefully these examples give a glimpse of how CIL would look to a user; we now address ourselves to the question of how CIL "works". This section shows how CIL tries to make sense of input commands and is hence an extension of section 4.2, while the next section is concerned with how CIL commands are evaluated.

The fundamental idea here is this: The GDBMS can make sense of something if it can assign one and only one entity ID to every name. If it cannot assign an entity ID to some name, the command is meaningless; if it can assign more than one name to some ID, the command is ambiguous. (If we humans make sense of something while taking one name to represent two entities, we are usually understanding a joke. As a GDBMS with a sense of humor is not our goal, this idea of meaning will be adequate.) Hence, we are really talking here about a name to entity ID mapping capability - in the same way as section 4.2 was talking (although not explicitly) about an entity ID to name capability. By making these processes explicit, we can view the GDBMS function as in Figure 5-2.

In order to make sense of CIL, we will be working here at the set level. It would seem that this is how people make sense of questions and statements. For instance, "What
Figure 5-2
are Joan's children called?" makes sense to look at, but it may well be meaningless because Joan doesn't have any children. On the other hand, we can see immediately that "What are that chair's children called?" does not make sense because, as we would put it, the child relationship is not defined for chairs. We are concerned at present with catching the second example but not the first, because the first only fails when we try to "evaluate" it to give the answer. To be more precise, we are concerned with finding the entity IDs, that names in a command can be mapped into, which make any relationships map between valid domain - range pairs.

The first problem is how to even go about looking for names, as the input is essentially only a string of character strings. Because of the way the syntax is set up, and because we have (and will) restrict relationship and set names to simple character strings (single words), it is possible to know the start and end of a name. From this, we can tell whether the name is a simple or compound character string. Furthermore, if the name is the direct object of a command, as ATLANTIS is in

UPDATE (GNP = $2) IN ATLANTIS,

the name must stand for the entity, rather than merely representing the character string ID. This is because the user should not be concerned with the CHARACTER STRING per se,
so she should never have to refer to the character string in this context. However, if the name is the value of a relationship in a predicate or fact, we will have to establish from the context if the name is the entity desired, or if it is merely the NAME of the entity represented. Hence, we will in general have to include both cases in the mapping process until one possibility is eliminated.

To show how the name to entity mapping works, we start with a simple example from section 1:

GET NUMBER OF PART ( ).

From the syntax we know that "PART" is the name of a set and that "NUMBER" is the name of a relationship which has that set as one of its domains. Hence the entity descriptions we seek will be of the form illustrated in Figure 5-3.

![Figure 5-3](image)

REP_SPEC here represents the ID of any repetition specification. Now it may be that there are two or more PART sets in
the data base; in this case, if NUMBER is defined on two of those sets, the command is ambiguous. If NUMBER is not defined on any set of the name PART, the command is clearly meaningless. Notice that ambiguity or meaninglessness of CIL commands is decided without evaluating the command; the command is "aimed at" the entity level, whereas the relationships are defined (and checked at this point) at the set level.

Also, notice that we did not entertain the possibility of two NUMBER relationships. We will take this to be a general rule; i.e., that a name cannot be the NAME of more than one relationship (ID). Now this doesn't exactly map reality, because some conceptually distinct relationships have the same name. For example, we may want to use PART as a relationship in the sense above, and in the sense of an actor having a part. However, for GDBMS usage, it will not make much sense to define conceptually different relationships of the same name on the same set. (Trying to remember which X is which in

```
DEFINE W
    X IS Y
    X IS Z, $END
```

would be hopelessly confusing). Hence, we will not create any ambiguity by using the same relationship ID for all relationships with the same name. So, for the following
discussion, we have two important restrictions on relationships:

1. Each relationship has a distinct, simple name.
2. For each relationship, there can only be one domain-range pair that specifies a particular set (and so, if a set is the domain of a relationship, we can unambiguously determine the range of the relationship).

As we can recognise a relationship from the syntax of a command, the first rule takes care of finding IDs for every relationship name.

This fact is a great help when it comes to mapping other names into IDs, especially in the case of identifying the set in an entity specification:

\[ \text{GET R1, R2 OF R3 (OF R4) OF S(R5 = V)} \]

If "S" is not a unique name of a set, the relationships will probably identify the correct set, for it must be the domain of the relationships:

1. R1 OF R4
2. R2 OF R3 OF R4
3. R5

Although the relationship names represent unique IDs, they may each have many domain-range pairs associated with them. Hence, S represents a unique set if one and only one set which has the name "S" is a valid domain for all the compound relationships defined on it. Figure 5-4 explains that
statement. The names of the relationships stand above the domain-range pairs they are defined on; S stands above representation of the IDs of sets named S. The dotted lines represent the identity of two IDs - so that D32, R42 and D12 all represent the same ID. The existence and uniqueness decision then becomes: is there exactly one tree of dotted lines that joins all the boxes (as the continuous line does)?

This is a complicated example. We hope that the normal state of affairs would be much simpler than this!

The previous two examples have been simplified by the information given by the position of the name - that the name represents a SET or RELN. In the last case, we have no such information - we are concerned with getting the IDs represented by entity names. These can either be the direct object of a command - as in the ATLANTIS examples of Chapter III - or they can be "values" of relationships in predicates or facts.

If the entity name represents the value of a relationship, we do have some information to narrow down which set it is in, because the value must be in the range of the relationship. Furthermore, if the entity specification refers to a set, that set must be the domain of the relationship which corresponds with the value's set as the range. As an example of this, Figure 5-5 shows the impli-
cations that "S (R5 = V)" have for the interpretation of "V". By finding the ENTITY-NAMED relationship for "V", we get the EN group of entities. Note that we have included "V" itself in this set; only after mapping to the set level can we decide whether "V" is the name of an entity or is just being itself. Next, we find the CONTAINING_SETs of this group, to get a group of sets, CS. Now, if the domain-range pair has been fixed ( (D52, R52) in the Figure), the set the named entity is in has been determined, which we hope would uniquely indicate the entity. (In this case it does - the entity named is V1. On the other hand, if (D53, R53) had been the indicate domain-range pair, either V2 or V4 could have been the entity represented by "V".) If the domain range pair has not been fixed, the set represented by "S" has not been fixed either; that is, there are two or more connected trees that join the relationships and the sets. Figure 5-5 then is an extension of Figure 5-4, and is thus a further restriction; there must be one tree connecting the relationships to the set and all the names to one entity to one containing set and thus to the relationships. In other words, the mapping of the set name to the set is in general dependent on the mapping of a value's name to the value, and vice versa.

Even if the name is a direct object of a command, there may be some contextual information - as in the example
already given:

**UPDATE (GNP = $2) IN ATLANTIS**

we know that ATLANTIS must be in a set that has a relationship GNP defined on it. In fact, there are only two situations where we will be without any indication of a name's set. These are commands like

**DESCRIBE ATLANTIS**, or

**DELETE ATLANTIS**.

In these cases, mapping is trivial; the command is accepted if "ATLANTIS" is the name of exactly one entity. If the name is not simple, the mapping requires a bit more work, which will be taken up in the next chapter.
5.3 Evaluating CIL.

In the general outlook involved in making sense of CIL is one to one mappings, our concern here will be with many to many mappings. This comes from two sources: the evaluation of an entity specification internally in the GDBMS yields a set (in the mathematical sense) of IDs; and the value of a repeatable relationship in an entity can also be a set of IDs.

An entity specification evaluation proceeds in two stages. First, the values for the relationships are evaluated; then the predicates are evaluated for each entity in the set (if one was specified, as will usually be the case; otherwise from the sets implied by the predicates). To illustrate this process, we will add the relationship EMPLOYEE REP to the SUPPLIER set defined previously, and consider the following (totally artificial) CIL command:

```
GET NAME OF SUPPLIER OF
  SUPPLY ( PART = ( GEAR, BOLT ),
            NUMBER OF PROJECT > 5,
            EMPLOYEE OF SUPPLIER SUPERSET (BILL,FRED))
```

which will retrieve the names of all suppliers who supply gears or bolts to any project whose number is greater than 5, and who employ both Bill and Fred. The evaluation of the "values" (the right hand sides of the predicates) yields three sets:
(GEAR, BOLT)
(5)
(BILL, FRED)

Now we evaluate the relationships in each predicate for all SUPPLY entities. Evaluating "PART" is trivial. Similarly "NUMBER OF PROJECT": we merely find the PROJECT of this SUPPLY entity, and then find the NUMBER of the PROJECT. Because these evaluations only involve SINGLE-REQ relationships, there will be exactly one for each SUPPLY. The EMPLOYEE relationship is RFPeatable, so the EMPLOYEE of SUPPLIER will yield a set of EMPLOYEE IDs. If we now represent the three values for a particular SUPPLY entity by

(P)
(N)
(E1, E2, ..., En)

then this SUPPLY is included in the specified set of entities if

1. P is GEAR or BOLT
2. N > 5, using normal arithmetic operations on the IDs.
3. For some i and j, Ei is BILL and Ej is FRED.

A word needs to be said about the "=" sign in predicates. We would like to use "=" in the case when we might get many IDs on both sides of it. So when is

(A1, A2, ..., An) = (B1, B2, ..., Bn)

true? It seems most useful to have it mean that one A is
the same as one \( R \). So if the third predicate in the previous example had been

\[ \text{EMPLOYEE OF SUPPLIER} = (\ \text{BILL}, \ \text{FRED}) \]

it would have been true for either Bill or Fred being employed by the SUPPLIER.

The only other evaluation of interest is evaluating a relationship of an entity specification. There are again two cases, of which the simpler is when the combined form is another entity specification. In example 3.7 in section 1, "SUPPLIER (NAME = JONES)" is an entity specification, but so is the "LOC OF SUPPLIER (NAME = JONES)". In this case, both entity specifications represent IDs of one SET, so the GDBMS can eliminate any redundancy in the resulting sets. For instance, "SUPPLY (COLOR OF PART = RED)" will represent a set of many SUPPLY IDs, whereas "COLOR OF SUPPLY (COLOR OF PART = RED)" will represent just one ID when all the redundancy has been eliminated.

The more complex case is in the evaluation of a GET command, when many relationships are to be evaluated over an entity specification. Examples 3.2, 3.10 and 3.11 from section 1 are examples of this situation. In this case, it will normally not be possible to eliminate any redundancy in the resulting values, because there could be different numbers of values from relationships: in 3.11, for each project ID the NUMBER and NAME relationships will yield one
ID value, but the last relationship yields any number. So in this case there is no point in trying to eliminate any redundancy; the value for the GET command will be the set of values of the relationships on each entity in the entity specification.

5.4 CIL in Perspective

In this section, we discuss the motivation for introducing CIL and also some of the advantages and disadvantages it has as a language.

The initial reason for developing CIL was to show that the entity view of information is very natural. The form of the entity specification is very similar to our box view of an entity, as shown in Figure 5-6. Clearly the meaning of these two forms is different. The entity description is a description of one entity, whereas an entity specification is a partial description of a set of entities. But the process of defining relational operations is the same at both the set and individual level — we merely apply a relationship to the entities (set or single) to obtain the values (set or single). This is analogous both to the human process of finding relational information, which starts with an (incomplete) description of a set of entities or a single entity and find the values of the relationships on
them, and also to the mathematical idiom - if \( f: A \rightarrow B \), and \( a \in A, b \in B \) then both \( F(a) \) and \( F(A) \) are valid notations, the first being a single object, the second a set.

\[
\begin{align*}
\langle \text{set-name}\rangle & \quad (R1 = ES1, \\
R2 = ES2, \\
\ldots & \\
\ldots & \\
Rn = ESn)
\end{align*}
\]

Entity specification

<table>
<thead>
<tr>
<th>Entity Description representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS / \langle \text{set-name}\rangle</td>
</tr>
<tr>
<td>R1 / V1</td>
</tr>
<tr>
<td>R2 / V2</td>
</tr>
<tr>
<td>\ldots</td>
</tr>
<tr>
<td>\ldots</td>
</tr>
<tr>
<td>Rn / Vn</td>
</tr>
</tbody>
</table>

**Figure 5-6**

However, our other reason was more important, and has little to do with the CIL in itself. The chapter has mainly been concerned with the GDBMS capabilities necessary to support a user language which is embedded in potentially ambiguous names. That is, every name used in communicating with the GDBMS can both be used in different ways and can also be the name of many entities. The reason for demonstrating CIL in such detail in this chapter was to be able to exhibit these capabilities.

So CIL and its supporting GDBMS function have been introduced as pedagogical devices; we will now reconsider these as practical tools, and their advantages and disadvan-
tages as such. In fact, most of our comments previously have been about the advantages of CIL, because they have also been our goals - a natural user view of her information structure and the freedom to use names in a human-orientated manner. Therefore, we will discuss the disadvantages.

The first is that, although CIL allows the user to talk about naturally structured information, it does so in an unnatural way. The user we want to extend the full power of CIL to is the non-programming user. Now it would seem very unlikely that a non-programmer could have constructed example 3.10 or 3.11 of section 1. Obviously one problem is that our syntax-nested parentheses do not reflect the person in the street's normal mode of expression. However, the deeper problem is that these examples express rather complex commands. So while dressing the CIL up in a more appealing form on the surface would be a desirable revision, I don't think it is going to make any particular command that much easier to express. (See Codd [Codd, 71] and Rothnie [Rothnie, 72] for alternative expressions of similar complexity).

The other disadvantage is not concerned with the CIL as such; rather, it is the whole approach we have taken to ambiguity in names. With current technology, our handling of names in a data-base of any size would be far too ineffi-
cient for a practical GDBMS. The normal strategy for making the name to entity mapping process viable is to define a relationship for each set which will uniquely identify an entity in the set – for example, giving parts unique numbers, or people unique names. The name to entity mapping in this case is one to one. The choice between these approaches is not clear cut; it is really a trade-off between efficiency, speed etc., and flexibility and ease of use, which naturally involves a subjective assignment of the relative importance of these factors, although as we have noted, at present the purely technical considerations outweigh these factors. (As a side note, a possible alternative strategy is the definition of "areas" (see [DBTG, 71, p. 13]). This is a method of partitioning the GDBMSs information into logically consistent subsets, which may overlap. Either a user defines an area as a collection of sets or partial sets which she declared she will primarily be referring to, or the GDBMS attempts to accomplish the same result by inference. The technique has the flavor of the working set approach to paging, and obviously one of the benefits: the area can be brought into a faster memory in the storage hierarchy and thus shorten access times. More pertinent to our naming approach here, the definition of an area would reduce the scope of name to entity mapping and hence reduce the probability of unintended ambiguity).
VI

PROCEDURES

6.1 Procedures and Virtual Information.

Until this point, our view of information has been limited to the usual one, in which the GDBMS is regarded as the handler of a set of given facts, and it is only responsible for the retrieval and storage of the facts which must be stored explicitly. On the other hand, in the classical view, a function such as square root or the trigonometric functions compute their outputs directly from their inputs with no other data needed.

In practice, of course, most interesting procedures fall between these two extremes, taking one or more inputs and needing files or other auxiliary information to produce their output. In this chapter, we want to introduce procedures into our model in this generalized form. Our approach is based on that of Folinus et al., which is outlined in the following quote:

One view of data-base systems is as a method of describing and mapping data structures into physical storage. An alternative view is that, given appropriate stored data, the problem is how we use it to meet requests for information. Requests for "answers", whether made to processing programs or a stored data base, are essentially requests for a value of a function, given various argument values.
A model of an information system as a collection of such functions helps unify many of our notions about data and algorithms, and provides a convenient construct for resolving several problems in data-base systems. [Folins et al., 74, p. 4]

The vehicle for introducing procedures is obviously the relationship. We have already noted in Chapter II the operational similarity of the relationship and a mathematical function - they both map from a member of one set to a member, or members, of another. Therefore, it is natural to allow some relationships to be defined procedurally rather than having their values stored explicitly in the data-base.

There are a few extra pieces of conceptual machinery we will need to handle procedures. The first is a language for specifying the procedures. We will not define such a language here, but merely indicate how the procedures would operate. The important thing about such a language is that it would contain all the power of the DML as well as the normal conditional, branching and arithmetic facilities, so that the procedures would relate to their information in a purely logical manner. Furthermore, a language of this sort would be conceptually simple to use and easy to understand, due to the fact that there would only be one "data type" - entity IDs. We will also need to store these procedures somewhere. Because the form of procedures is
different from the form of facts or of the contents of the translation data-base, we postulate a third data-base for storing the procedure definitions. Finally, we will need more set definitions. To be consistent, there will need to be a set for the procedures, but it will only be used as a range in any set definitions because all the information about the procedures themselves will be stored in the procedure data-base. We will also need to revise the relationships set definition, as follows:

```
DEFINE RELN,
    NAME IS CHARACTER_STRING,
    DR_PAIR REQ,
    FORM IS CHOICE (EXPLICIT, VIRTUAL) REQ,
    PROCEDURE REP,
$END
```

The FORM relationship is the indicator of which way the relationship can be evaluated; we do not exclude the possibility that a relationship can be evaluated both ways. The PROCEDURE relationship will give the IDs of any procedures that can be used to evaluate the relationship.

The responsibility for recognizing the need to use a procedure and for actually using it falls to the "evaluation" module; so we can recast Figure 5-2 as Figure 6-1. With this structure, once a procedure has been defined for a relationship, it can be used in exactly the same way as
we have been accustomed to. In this way, we have "virtual information", to use the term of the Polinus et al. paper; however, the value of the relationship is arrived at, it appears as though the value is stored explicitly.

For example, the simplest use of procedures would be to store the definitions of compound relationships, such as grandmother, uncle etc. Instead of storing the values of such relationships explicitly, the relationships could be defined along with the simple procedure to evaluate them. Another use along similar lines would be to define the familial relationships in their basic parent, child, sibling form initially; and from them to define the other relationships via procedures. So for instance the SON procedure would find the children of its argument ID, and from them extract those with \( \text{SEX} = \text{MALE} \).

Another example from the Polinus et al. paper is that
of the AGE relationship. This is an example of a relationship that is by nature time dependent. The only way to keep it accurately valued without enormous overhead is to recompute its value with each request — as current data minus date of birth. Strongly related to this are the (probably system) functions like COUNT, TOTAL etc, which are again essentially time dependent. If these are implemented as virtual relationships, they become rather general operations. For they are mappings from a set of entities to one entity, and thus are applicable to any entity specification as well as to defined SETs, as long as the entity specification yields entities in the domain of the function. That is, using the sets defined in section 1 of Chapter V,

\[
\text{GET TOTAL OF QUANTITY OF SUPPLY(\textit{NAME OF SUPPLIER} = \textit{JONES})}
\]

or \[
\text{GET COUNT OF PART(\textit{QOH} < 25)}
\]
can be evaluated by the same sort of procedure as would be used to evaluate

\[
\text{GET COUNT OF PROJECT ( )},
\]

6.2 Procedures as Knowledge.

One of the hopes behind the GDRMS slogan is that an information handling facility can appear more intelligent than its present day counterparts. Now intelligence in any
particular situation is based on the "knowledge" of what the situation as all about. For instance, if a GDBMS is to react intelligently to a request for information about the children of a person when the CHILD:relation is not stored explicitly, it must "know" that CHILD is the inverse of PARENT. As computer systems are not susceptible to mystic revelations, any knowledge they may have must at some point have been programmed into them.

In this section we are concerned with making the GDBMS sophisticated about consistency checking and protection via procedures. We have already encountered one non-procedural consistency check in repetition specifications, and within our already existing framework it is clearly possible to set up a permission list mechanism for protection. The motivation for a procedural approach to these areas is the desire for more flexibility, to make the GDBMS appear more intelligent. For example, in the definition of RELN given in the preceding section, it makes little sense to have $\text{FORM} = \text{VIRTUAL}$ if no procedure is given by the PROCEDURE relationship; a consistency checking procedure for the RELN set would be needed to handle the FORM relationship automatically and to make sure it was consistent with the procedure information specified.

However, we are not particularly concerned with the effect of the procedures. It seems that there are so many
consistency or protection procedures possible that it would be inefficient and futile to try and incorporate them all into the GDBMS structure. What we should be concerned with is making the procedures easy to insert into the system structure. One element of this is obviously an easy to use and understand language; because a host language would have the DML as a subset, all manipulation would be done in a data structure independent environment, and so the procedures should much more nearly reflect the inherent logic of the job being done (the actual knowledge) than current programming languages. Another factor in whether these procedures are easy to insert is the complexity of the environment in which they must operate - whether or not it takes a major intellectual effort to understand the environment; we can only hope that the GDBMS environment is amenable to comprehension by the non-professional more than most current operating systems are.

The ease of insertion factor we are really interested in here is the nature of the "hook" by which the procedures are attached to the GDBMS structure. This hook can be very simple in our structure because the procedures can be made the values of certain system-defined relationships, and then at a certain point in the GDBMS' operation be "called" automatically by virtue of being the value of that relationship. This facility should be easy to use because there seem to be a limited number
of situations in which this sort of procedure would be invoked.

There are two dimensions to these situations. One is the "level" of the procedure, of which there are only three: the set level, to check the operation for any member of the set; at the relationship level, to check operations pertaining only to a particular relationship; and for particular entities, to check operations affecting that one entity. The other dimension is the type of operation being performed: these can be broken down into read, create, update and delete. This gives only twelve types of checking procedure for either consistency checking or protection, and then some of them can be discounted. For instance, there would be no use for a procedure to check the consistency of a read operation because the read operation does not affect the state of the information. Another example is that it doesn't make sense to have a particular entity procedure checking the creation of the entity, because the entity description must exist before the procedure could be entered into its description! The entity set definition to handle this facility would then look like this:

```
DEFINE ENTITY,
    CONTAINING_SET IS SET,
    READ_PROTECTION_PROC IS PROCEDURE,
    UPDATE_PROTECTION_PROC IS PROCEDURE,
    DELETE_PROTECTION_PROC IS PROCEDURE,
```
and the definitions of SET and RELN would be updated similarly.

One example of the use of these procedures is in allowing redundant information into the system. This capability would clearly cost more in storage costs, but at some point may be viable because of the benefits of decreased access time, and the greater possibility of restoring system information after a crash. This is a situation that would be handled by the checking procedures for updating, creation and deletion at the relationship level. The most obvious case of redundancy is storing a fact in its inverse form as well as its normal form. For example, a set definition would contain a MEMBER relationship, giving all the members in that set, as well as having the CONTAINING_SET relationship defined in the entity. Another example is storing both PARENT and CHILD relationships explicitly. These are rather simple examples, and could probably be handled by a "static" definition of the inverse relationship. However, this basic mechanism is able to handle arbitrarily defined spontaneous updating (similar to the TRW systems GIM correlative, as described in [Diebold, 69], which is not easily amenable to non-procedural description.

Another use of these ideas is in handling the translation procedures for non-simple names, which we referred to in the
previous chapter. The insertion of these procedures into the data-base would follow exactly the same lines as above, by defining IN_TRANSLATION_PROC and OUT_TRANSLATION_PROC relationships at the set level. These procedures would be of a different nature than the above checking procedures, which can operate within the existing information structure. The translation on the other hand would have to deal with the linear ordering of entity IDs, which is outside the scope of our structuring. For instance, translation an ADDRESS into its "name" form would involve ordering the values of the ADDRESS's relationships into a well-defined sequence, such as NUMBER, STREET, CITY etc. It will clearly require an extension of the host language to handle this, but the interaction between the procedure and the evaluation module would be the same as before.

The advantage of this approach is that it shifts the responsibility for specifying the particular knowledge required in the GDBMS onto the user of the GDBMS and away from the GDBMS itself. This is a desirable approach for both parties. The GDBMS does not need to be concerned with the particular special facilities required; as long as the necessary hooks are provided, the use of the facilities can be supported by a simple table-driven system. On the other hand, the system becomes accessible to the users, and hence more easily specializable to the particular functions desired than if the procedures incorporating all the possible special facilities were bound in an ad hoc fashion into a monolithic system.
VII

CONCLUSION

In conclusion we examine the relation of some aspects of this thesis to other work in the GDBMS area.

7.1 Assumptions.

We have made some assumptions that to a large extent have guided the development given, which we must discuss at this point.

The first is that it is possible to talk about a purely logical information structure independent of any considerations of the physical representation or communication of that information. The validity of this assumption is of course a function of the type of information under consideration: it would fall down in an examination of the totality of human information, but for the subset of information we have discussed it appears to be viable.

Following on from this is the assumption that, with regard to this subset of information, there is no conceptual difference between the human "GDBMS" and a computer GDBMS. We have implicitly used this assumption repeatedly by mapping our human view of information into the computer idiom and claiming that is how the computer GDBMS should be structured. This
approach was particularly evident in our handling of names. Even if the assumption is invalid in some cases, the inescapable fact that GDBMSs have to interface with humans should make this a worthwhile approach to understand the facilities that a GDBMS should provide.

Our other two assumptions have to do with the way the GDBMS is used. The typical information system is mainly concerned with large amounts of information that is fairly simply structured in large sets and which is valuable as a whole but not particularly valuable at a microscopic level. We have assumed a single interactive user asking in general for rather small amounts of information, and also that the value of having that small amount of information is much higher than the value of a similar quantity of information in current systems. This is essentially the same assumption as the philosophy behind the MIS; the manager needs to be relieved of the burden of extracting the information he needs from a boxful of computer printout, and instead be given a smaller quantity of data with a much higher information content. Because of these assumptions, we have ignored the transfer of large amounts of information, and the interface with "normal" programs; however, these are clearly important elements of a real GDBMS.

7.2 Other Theoretical Work.

In order to discuss the relationship of this work to
others in the field, we will briefly review the main ideas presented.

Our starting point was the introduction of the three realms involved with information: real world entities, stored information about them and the communication of information. We then make a strong separation between three broad GDBMS functions: the storage and accessing of information; the structuring of the information; and its communication. Because of this separation, it was possible in considering the structure of information to deal exclusively with entity IDs, which are abstract objects that uniquely identify real world entities. Our basic element of information was the fact, which is the basis for two structuring processes. The first structuring is the description of an entity, which is a conceptual gathering together of all facts that describe the entity; the second is the templating process of the set level. The set level structure is self-consistent in that it can define itself. We next discussed the problem of effective communication, which was defined as being able to uniquely identify the entity being referred to in a potentially ambiguous user language. The fundamental thesis was that a clear conceptual separation of the following was needed: entity IDs, the name(s) of an entity, internal IDs of character strings, and their external form. Finally, Chapter 6 incorporated procedures into the GDBMS, which potentially allowed the kernel of the GDBMS at the
information structuring level to be greatly simplified by removing procedural knowledge from its domain of responsibility.

The forerunners of this work which we shall now discuss are those of Mealy [Mealy, 67], Codd [Codd, 70 and 71], Engles [Engles, 72] and Senko et al. [Senko et al., 73, Astrahan et al, 72]. Codd's work is a complete theoretical development of the relational view of information as originally proposed by Mealy, and is fundamentally different from the view taken here. The benefit of the relational view is that it allows the application of powerful mathematical tools to a table structuring of data. The table structuring is essentially no different from the classical view of a file as an array, which explains its immediate appeal. The main drawback is that it has very little connection with the natural structure of information - we just don't think of information in table terms, we relate to facts about particular objects - although of course the systems can be made equivalent. This drawback is particularly evident in the cumbersome normalization procedure required to handle a domain in a relation whose elements are themselves relations; in our model, this case is simply an instance of a many-valued relationship.

Another idea introduced by Mealy is that of non-entity sets, and "non-structural" data maps from entity sets onto non-entity sets. Engles has also further divided "structural" maps into maps from a set onto itself, and maps from one set onto
a different one. Operationally or pragmatically this may be
a valid distinction. However, it does mask the fundamental
unity we have tried to show in this area: there is conceptually
one class of objects we are interested in, namely entities
(everything is an entity); and there is only one type of data
map, what we have called a relationship. Doing away with the
unnecessary distinctions allows a very simple and homogenous
information structure model, which is surely desirable at least
as a conceptual aid if nothing more.

Our approach also differs from that of Mealy and Engles
in our definition of the partinent "realms" in information
modeling [Mealy, 67, p.529; and Engles 72, p.13]. Engles' view
is that "The three realms are: the real world, ideas about it
existing in the minds of men, and symbols on paper or some
other storage medium." [Engles, 72, p.14] We have made these
last two into essentially one realm, because of our view that
the essence of human ideas of information should be captured
in the information structure of the GDBMS. Also, we have chosen
to consider the communication process as another realm because
of its importance for information in general, and because it
affects the content of the data-base. However, the difference
here seems to be more one of taste than one of perception.

As we have already noted, this work is heavily based on
that of Senko et al. There are many small differences with their
work, which are again a matter of taste, and two major ones.
The first is illustrated by the very name of his model—The Entity Set Model. And indeed, the emphasis in the paper is on the structuring of the Entities by the definition of the Entity Set. In contrast, our approach could be called the "entity and set" model, for two reasons. One is that we explicitly allow an entity to be in more than one set; Senko seems to leave this dependence undefined. This allows full recognition of the independence of the entity and set concepts. The other reason is our emphasis on the importance of the entity description as a separate level of abstraction in the model, which is a better representation of our natural thought processes. As Senko et al. themselves say, "Information consists of facts about things [entities]." [Senko et al., 73, p. 46]

Our second major difference with Senko's work is our handling of names. His approach is outlined in the continuation of the previous quote:

"These facts and things exist independently of any representation, but it is essentially impossible to deal with them conceptually except in terms of some representation. As soon as we draw a picture of something or give it a name, we are dealing with the thing in terms of a representation. In a data-base system, we deal with information almost completely in terms of name representations of information." [Senko et al., 73, p. 46]

As a result, the development is couched entirely in terms of names, which becomes very cumbersome. Our entity ID is precisely an attempt to deal with facts and entities in a manner
completely independent of any representation. Our approach is simpler because of the distinction between internal character string IDs and their external representations, and because of our explicit use of the entity ID, which has nothing to do with names. Once again, this is meant to more accurately reflect our psychological experience.

7.3 The Real World.

The whole development until now has been theoretical, apart from the brief discussion of the cost of our approach to naming in Chapter V. To conclude, we make a brief excursion into a more concrete sphere to discuss the implication of real world constraints on GDBMS construction for our model. There seem to be five important dimensions to the constraints on a GDBMS:

1. Ease of use.
2. Programming cost.
3. Speed of operation.
4. Cost of operation.
5. Storage space.

Of course, we have been implicitly concerned with the first constraint throughout the thesis. By attempting to faithfully model the human's relation to her information, we have hopefully shown an easy to use structure that the GDBMS could present to the user. As with all interfaces, the structure of
the user interface is essentially independent of the way it is simulated.

Rather, the question we want to address here is whether the information structure we have shown is viable as a method of GDBMS construction. That is, is it possible to construct a GDBMS with such a strong separation between the storage of the information and the manipulation of it that this intermediate manipulation level could operate in a data-less manner — without tables of its own? In terms of the above constraints, the simplicity of the middle level would make programming it simpler. However, as it is, for each request the intermediate level would need information from the physical storage modules on each entity, set and relationship involved in the request. With present technology, any reasonably complex request to the GDBMS would become immensely costly and would be answered too slowly. Hence, this method of construction would be infeasible.

There are two alternatives to this approach. The first is the normal method of construction, in which the intermediate level keeps set and relationship information and so can access it much more cheaply. The other approach is to support the physical storage routines with hardware associative memories. [For example, see DeFiore and Berra, 74] This would allow localised references to be handled very quickly by allowing the intermediate level to use this associative memory directly, and so by-pass the storage accessing routines of the physical
level, which would probably be rather complex and time-
consuming in a GDBMS of any size. This approach is of course
dependent on the realization of viable associative memories
of sufficient size.
APPENDIX I

MEMORY INTERFACE

The super-memory interface is specified here as a set of PL/I subroutine calls. Although the orientation is towards an implementation, the details and exact form of the calls are of course unimportant; the purpose is to make the nature of the interface more concrete, which is best done in a programming language. The calls are written as complete statements. NULLID is the ID that is not an ID of any entity, analogous to the NULL pointer of PL/I.

1. Addition:

   a) For a single fact:

      BOOLEAN = ADDFACT( ID1, RELN, ID2 );

      The returned value indicates whether the fact existed prior to the call.

   b) To add many values:

      CALL ADVALUES( ID1, ID2, ID3, POINTER );

      One of the IDs must be NULLID; the set of IDs to be inserted in its place is in the list pointed to by POINTER.

   c) To pairs of values:

      CALL ADD_DOUBLE( ID1, ID2, ID3, POINTER );

      Two of the IDs must be NULLID; the pairs of IDs to be inserted in their place is in the list pointed to by
POINTER. This call for example could be used to insert the description of a new entity.

2. Deletion:

All deletions take the following form:

```plaintext
BOOLEAN = DELETE( ID1, ID2, ID3 );
```

One or two of the IDs can be NULLID, in which case any ID can match the NULLID position. The returned value indicates whether any facts were deleted as a result of the call.

3. Query:

a) To find if a fact or partial facts exist:

```plaintext
BOOLEAN = EXISTENCE( ID1, ID2, ID3 );
```

Again, one or two of the IDs may be NULLID. The returned value indicates whether any facts were found to fit the specified IDs.

b) To retrieve values for two IDs:

```plaintext
POINTER = GET_VALUES( ID1, ID2, ID3 );
```

One of the IDs must be NULLID; the list of IDs that make a fact when inserted into the NULLID position is pointed to by POINTER.

c) To retrieve pairs of IDs:

```plaintext
POINTER = GET_DOUBLE( ID1, ID2, ID3 );
```

Two of the IDs are NULLID, and the list of pairs of IDs that make a fact when inserted into those positions is pointed to by POINTER.

d) To find out how many facts fit a specification:
INTEGER = GET#( ID1, ID2, ID3 );

One, two or three IDs may be given.

4. ID Operations:

   a) To ask for a new ID:

      IDX = NEWID;

      The new ID is assigned to IDX.

   b) To purge an ID from the memory:

      CALL DELETE( ID );

      The ID, its description and all facts it appears in are deleted.
APPENDIX 2.

BNF DEFINITION OF CIL

In this definition we will use the following "meta-PNF" item:

```
<constructs> ::= <construct> [, <construct> ] ... 
```

```
<CIL command> ::= <new> | <add> | <delete> | <update>
                  | <get> | <true> | <describe>
```

```
<new> ::= NEW <entity descr.>
```

```
<add> ::= ADD ( <facts> ) TO <entity spec>
```

```
<delete> ::= DELETE {<entity spec> |
                  <reln names> FROM <entity spec> | ( <facts> ) FROM <entity spec>}
```

```
<update> ::= UPDATE ( <facts> ) IN <entity spec>
```

```
<get> ::= GET <relns> OF <entity spec>
```

```
<true> ::= TRUE ( <facts> ) IN <entity spec>
```

```
<describe> ::= DESCRIBE <entity spec>
```

```
<entity descr.> ::= [<set names>] ( | <facts>.)
```

```
<fact> ::= <reln name>= {<new> | <entity spec>}
```

```
<entity spec> ::= [<reln> OF]...[<set names>] (|<predicates>])
```

```
<predicate> ::= <reln> <op> { <entity spec> | ( <entity specs> ) } 
```

```
<reln> ::= <reln name> | <compound reln> | <reln defn>
```

```
<compound reln> ::= <reln> OF <reln>
```

-106-
\(<\text{reln defn}\> ::= (\text{RELN} (\langle\text{ID name}\rangle) \langle\text{relns}\rangle \text{ OF } \langle\text{entity spec}\rangle )

\langle\text{op}\rangle ::= < | \leq | = | \geq | > | \text{SUBSET} | \text{SETEQ} | \text{SUPERSET}
APPENDIX 3

BNF DEFINITION OF SDL

We once again use the "meta-BNF" item:

<constructs> ::= <construct> [, <construct> ]

<new set defn> ::= DEFINE <new set name>[IS <superset defn>],
                     <reln defns>,
                     [ATTACH ( <facts> )],
                     $END

<superset defn> ::= <set name> [AND <superset defn>]

<reln defn> ::= {<reln name> IS <set defn> | <set defn>}
               [[<repetition spec>] [<form spec>]

<set defn> ::= <new set defn> | <old set name>

<repetition spec> ::= SINGLE_REQ | SINGLE | REQ | REP

<form spec> ::= EXPLICIT | VIRTUAL

Note: <facts> is a CIL construct, and is defined in
Appendix 2.


