1975

A Model of Data Structures Commonly Used in Programming Languages and Data Base Management Systems

William L. Honig
Loyola University Chicago, whonig@luc.edu

Recommended Citation
A MODEL OF DATA STRUCTURES COMMONLY USED IN PROGRAMMING
LANGUAGES AND DATA BASE MANAGEMENT SYSTEMS

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY
Field of Computer Science

By

WILLIAM LEONARD HONIG

Evanston, Illinois
August 1975
to

LINDA

who defines structured programming as "programming for the future"
ABSTRACT

This thesis claims that contemporary data structures can be understood and studied with an intelligible model which captures their essential differences and similarities and, further, that such a model is an appropriate basis for a top-down description method for data structures. To define the scope of the model, the data structures included in 21 programming languages and data base management systems have been tabulated. Each individual data structure is illustrated with an example drawn from a published paper or a working computer program. This mélange of data structures is divided into three classes (aggregates, associations, and files) and each class is modeled with a set of questions. Each question delineates one significant characteristic of the data structure and can be viewed as one axis of an n-dimensional universe of data structures. To demonstrate the clarity and generality of the model numerous existing examples, including several CODASYL Data Base Task Group and Feature Analysis data organizations, are described with the model. Additionally, a "completeness" exercise demonstrates that the model can represent all of the data structures identified in the survey of 21 programming languages and data base management systems.

The top-down data structure design method is based upon the model and is particularly suited to both the design and documentation of large data bases. Two special features, restatement and redefinition, allow the designs to remain intellectually manageable throughout a large number of conceptual levels. To show the utility of these methods a practical data base design for a software development system is presented. The requirements for this data base are drawn from the typical situation in which a number of
individual programmers cooperate to create a software system which is used and modified over a long time period. This design proffers a general solution to a common programming problem and is thus a "software engineering" approach to data base design.

In order to compare and contrast the model of this thesis with existing work, 11 other data structure models are surveyed and divided into four groups: semantic, prototype, analysis, and information models. The data structure model of this thesis is an analysis model; such models provide a compilation of all possible variations among a collection of data structures. To aid the comparison, a common example is expressed in terms of each data structure model.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>1</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>11</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>xi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Organization</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Example Data Structures</td>
<td>5</td>
</tr>
<tr>
<td>2. COMMONLY USED DATA STRUCTURES</td>
<td>22</td>
</tr>
<tr>
<td>2.1 General Trends in Data Description</td>
<td>25</td>
</tr>
<tr>
<td>2.2 Common Data Structures</td>
<td>29</td>
</tr>
<tr>
<td>2.2.1 Basic Items</td>
<td>33</td>
</tr>
<tr>
<td>2.2.1.1 Storage</td>
<td>33</td>
</tr>
<tr>
<td>2.2.1.2 Pointer</td>
<td>35</td>
</tr>
<tr>
<td>2.2.1.3 Conceptual Items</td>
<td>38</td>
</tr>
<tr>
<td>2.2.1.4 Equivalence</td>
<td>57</td>
</tr>
<tr>
<td>2.2.2 Aggregates</td>
<td>59</td>
</tr>
<tr>
<td>2.2.2.1 Table</td>
<td>62</td>
</tr>
<tr>
<td>2.2.2.2 Group</td>
<td>82</td>
</tr>
<tr>
<td>2.2.3 Associations</td>
<td>94</td>
</tr>
<tr>
<td>2.2.3.1 Pointer Based Association</td>
<td>94</td>
</tr>
<tr>
<td>2.2.3.2 Other Associations</td>
<td>104</td>
</tr>
<tr>
<td>2.2.4 Files</td>
<td>108</td>
</tr>
<tr>
<td>2.2.4.1 Arbitrary Access Algorithm</td>
<td>108</td>
</tr>
<tr>
<td>2.2.4.2 Indexing</td>
<td>109</td>
</tr>
<tr>
<td>2.3 Observations</td>
<td>118</td>
</tr>
</tbody>
</table>
## 6. ARGUMENTS FOR UTILITY

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Completeness</td>
<td>337</td>
</tr>
<tr>
<td>6.2 Usefulness for Top-Down Design</td>
<td>340</td>
</tr>
<tr>
<td>6.3 Ease of Use</td>
<td>346</td>
</tr>
</tbody>
</table>

## 7. CONCLUSION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Further Work</td>
<td>351</td>
</tr>
<tr>
<td>7.2 Contributions</td>
<td>359</td>
</tr>
</tbody>
</table>

## APPENDIXES

| A. Table of Data Structures in 21 Common Programming Languages and Data Base Management Systems | 364  |
| B. Completeness Exercise                      | 372  |
| C. Entire Software Development Data Base      | 382  |
| D. Glossary of Terms                          | 396  |

## REFERENCES

| REFERENCES                                   | 399  |

## VITA

<p>| VITA                                         | 410  |</p>
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Organization data base</td>
<td>7</td>
</tr>
<tr>
<td>1-2</td>
<td>Elements of organization data base, part 1</td>
<td>8</td>
</tr>
<tr>
<td>1-3</td>
<td>Elements of organization data base, part 2</td>
<td>9</td>
</tr>
<tr>
<td>1-4</td>
<td>Elements of organization data base, part 3</td>
<td>10</td>
</tr>
<tr>
<td>1-5</td>
<td>Decision table for LOAD instruction</td>
<td>12</td>
</tr>
<tr>
<td>1-6</td>
<td>Traditional decision table</td>
<td>14</td>
</tr>
<tr>
<td>1-7</td>
<td>Modified traditional decision table</td>
<td>15</td>
</tr>
<tr>
<td>1-8</td>
<td>Execution of decision table</td>
<td>17</td>
</tr>
<tr>
<td>1-9</td>
<td>Scheduling data base</td>
<td>19</td>
</tr>
<tr>
<td>2-1</td>
<td>Data independence DBMS</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>ELL array and matrix</td>
<td>32</td>
</tr>
<tr>
<td>2-3</td>
<td>Comparison of terms</td>
<td>34</td>
</tr>
<tr>
<td>2-4</td>
<td>Storage</td>
<td>36</td>
</tr>
<tr>
<td>2-5</td>
<td>General address pointer</td>
<td>37</td>
</tr>
<tr>
<td>2-6</td>
<td>Type qualified pointer</td>
<td>39</td>
</tr>
<tr>
<td>2-7</td>
<td>Typeless and typed language declarations</td>
<td>41</td>
</tr>
<tr>
<td>2-8</td>
<td>Type construction</td>
<td>43</td>
</tr>
<tr>
<td>2-9</td>
<td>User defined types with and without names</td>
<td>44</td>
</tr>
<tr>
<td>2-10</td>
<td>Regular numbers</td>
<td>46</td>
</tr>
<tr>
<td>2-11</td>
<td>Tabular</td>
<td>47</td>
</tr>
<tr>
<td>2-12</td>
<td>Boolean</td>
<td>48</td>
</tr>
<tr>
<td>2-13</td>
<td>Picture</td>
<td>49</td>
</tr>
<tr>
<td>2-14</td>
<td>Constant</td>
<td>51</td>
</tr>
<tr>
<td>2-15</td>
<td>Special data</td>
<td>52</td>
</tr>
<tr>
<td>2-16</td>
<td>Fixed length string</td>
<td>53</td>
</tr>
<tr>
<td>2-17</td>
<td>Variable length string</td>
<td>55</td>
</tr>
<tr>
<td>2-18</td>
<td>Functional virtual item</td>
<td>56</td>
</tr>
<tr>
<td>2-19</td>
<td>Elsewhere virtual item</td>
<td>58</td>
</tr>
<tr>
<td>2-20</td>
<td>Two things at same time equivalence</td>
<td>60</td>
</tr>
<tr>
<td>2-21</td>
<td>Equivalence of alternatives</td>
<td>61</td>
</tr>
<tr>
<td>2-22</td>
<td>Array</td>
<td>63</td>
</tr>
<tr>
<td>2-23</td>
<td>FORTRAN loop accessing array</td>
<td>64</td>
</tr>
<tr>
<td>2-24</td>
<td>PASCAL SALHIST array with tabular index</td>
<td>66</td>
</tr>
<tr>
<td>2-25</td>
<td>Matrix</td>
<td>67</td>
</tr>
<tr>
<td>2-26</td>
<td>PASCAL MSALH matrix with tabular indexes</td>
<td>69</td>
</tr>
<tr>
<td>2-27</td>
<td>FORTRAN ad hoc structure using matrix</td>
<td>70</td>
</tr>
<tr>
<td>2-28</td>
<td>Set</td>
<td>72</td>
</tr>
<tr>
<td>2-29</td>
<td>N-tuple</td>
<td>72</td>
</tr>
</tbody>
</table>
2-30 MADCAP VI n-tuple for a range
2-31 Sequence
2-32 N-ary relation
2-33 Relation with two identical domains
2-34 Binary relation
2-35 Hierarchy
2-36 Nested hierarchy
2-37 Additional declarations of nested hierarchy
2-38 Hierarchy with alternatives
2-39 Repeating structure
2-40 Organization database redrawn
2-41 Data definition for redrawn organization
data base
2-42 Repeating structure with count field
2-43 Tree
2-44 VDL tree object
2-45 Linked list
2-46 Directed graph
2-47 Owner-member structure
2-48 Functional association
2-49 Key file
2-50 Special key file
2-51 Current pointer file
2-52 Sequential file

3-1 FORTRAN Array
3-2 VERS2 Array
3-3 PASCAL Array
3-4 Model for aggregates
3-5 Model for array of Figure 3-1
3-6 Model for one row of decision table
3-7 Model for data structure of Figure 3-2
3-8 Model for entire decision table
3-9 Model for data structure of Figure 3-3
3-10 Model for set of conditions from decision
table
3-11 Model for array of nonhomogeneous elements
3-12 Use of "Number" axis
3-13 Model for all aggregates
3-14 Model for SKILLS relation
3-15 Truth table approach to data aggregates
3-16 Association data definition
3-17 Example association
3-18 Table form of association instances from
Figure 3-17
Top-down decision table design continued

Scheduling data base level 0

Scheduling data base level 1

Scheduling data base level 2

Scheduling data base level 3

Scheduling data base level 4

Scheduling data base level 5

Scheduling data base level 6

Scheduling data base level 7

Scheduling data base level 8

Alternative scheduling data base

Three models of array

Classification of data structure models

Two structure models of array

Other work on data structure models

Common example data structure

Liskov and Zilles' language for axiomatic model

Shneiderman's Structured Data Structure

Earley's VERS

Fleck's list structure

Earley's VERS2

Hsiao and Harary's generalized file

Smith's Generalized Data Description Language

Codd's relational model

Senko's Entity Set Model

Bachman's Data Structure Diagram

Redefinition and restatement
I wish to acknowledge, with gratitude, the support, willingness, and devotion of all the members of my committee. I especially thank Professor Mittman for his general sagacity and unwillingness to accept any lacunae in my writing and Professor Carlson for being a refreshing guide to relevant information throughout computer science and for using my data structure model in the classroom.

I am also grateful to Bell Laboratories for financial support under the Doctoral Support Plan, to my supervision whose recommendations made the support possible, and to the people and projects of Bell Laboratories for providing the practical computer science environment to counterbalance my academic studies. Special thanks is due Steve Campbell for helping to determine the requirements for the software development data base. Finally, I deeply appreciate the work of Rose, Sue V., Debra, Rita, Joan H. and the others without whose MT/ST expertise this thesis could never have been produced.
1. INTRODUCTION

This thesis claims that contemporary data structures can be understood with an intelligible model which captures their essential differences and similarities and, further, that such a model is an appropriate basis for a top-down description method for data structures. This introductory section discusses briefly the motivation behind the work, presents the overall organization of the thesis, and describes some example data structures for use in the sequel.

1.1 Motivation

Data structures are a vital topic in many areas of computer science. Because of this emphasis there has been a proliferation of data organization techniques and names for them. This situation might be called a "terminology overload;" it is common for different programming systems or textbooks to use the same name for different data structures and to use unlike names for the same basic data organization.

What is needed is an understanding of the basic characteristics of data structures - a means to unveil the differences and similarities among different data structures. The data structure model proffered here describes data structures in a way which is not dependent upon names for various data organizations. Instead, a model which makes clear the "parameters" or "degrees of freedom" for each of several classes of similar data structures is defined. With this model it is possible to represent a particular data structure without assigning a (new or old) name to it, but by specifying a set of proper parameter values.

Thus, this data structure model remedies the confusion about data structure names. In addition, this model is used
as the basis of a unique method for the top-down design and documentation of data structures. This top-down data structure description method contains some original features which seem particularly appropriate for the top-down description of large data bases.

The currently popular ideas of structured programming have spawned quite a debate over ways to design and describe data. [Gries 1974, p. 657], in a listing of current research into areas which might be called structured programming, states "research in this area of sp [program notation] is devoted to improving notation: ... (3) by learning how to describe data structures in a cleaner fashion." This quote describes very well the need which the data structure model and top-down design method are intended to fill.

The approach or direction of the research reported here has been influenced by the following philosophies:

1. The structural aspects of data organization can be studied independently from considerations of data access,

2. A data structure model should not impose too much structure, i.e., restrictions or implications which exceed the needs of a certain task should not be required,

3. A top-down design method should control details so that the design can be presented in an intelligible fashion, and

4. The data structures from programming languages and data base management systems should be studied jointly.
These philosophies portend the approach of the following work. First, the data structure model contends only with the static, time-invariant, structural aspects of data organization. The vagaries of accessing and changing a data structure are beyond the model's scope. This approach, while restrictive, does facilitate the understanding of a wide variety of common data structures. The reader may feel that some parts of the data structure model (e.g., the "Identification" axis of the aggregate model which tells how a component is named or labeled) infringe on the realm of access. Indeed, there can be no clear line between structure and access; the data structure model considers characteristics which primarily reflect structure. Additionally, some parts of the model provide insights into data integrity which, in some sense, lies on the middle ground between structure and access.

The second philosophy arises from the common confusion of imposing too much organization on a data structure. For example, there is no need for an array to be thought of as stored contiguously in memory. Likewise, a set should be able to exist in a form such that very little is known about its internal organization. A set may be manipulated with union, intersection, and member-test operations quite independently of any particular order among its elements. The data structure model developed here encourages clearer, more precise thinking about what exactly is required of a particular data structuring technique.

The third philosophy represents the belief that the primary goal of top-down design is the understandability or "intellectual manageability" of whatever is being designed. A top-down description of a data base should, thus, present the features and details of the data structures in a way which is easy to understand and master.
Finally, the data structures which the model covers exist in a large variety of programming languages and database management systems. The success of the data structure model in describing this collection shows there is no longer any need to consider the two sources separately, as far as data structures are concerned.

1.2 Organization

This thesis is divided into seven main sections plus appendixes. Section 2 surveys the mélange of data structures offered by programming languages and database management systems; it is, in effect, a guided tour of the large chart presented in Appendix A. This appendix, which represents a major input to this thesis, records the data structures provided by each of 21 programming languages and database management systems. The 21 systems covered were chosen to include all common data structures from popular programming languages and database management systems. Thus, while this set of 21 is not meant to be an exhaustive collection of all popular systems, it is representative of all the popular data structuring techniques. Section 2 describes, as briefly as practicable, each of the data structures listed on the left side of Appendix A's charts.

Section 3 introduces and develops the data structure model and demonstrates its use. The data structure model is the heart of this thesis; the remaining sections extend it, document its value, and compare it to other work. Section 4 describes a top-down approach to data structure description. This top-down approach is based on the data structure model and also includes two new top-down features called "restatement" and "redefinition." The data structure model and the top-down design methodology together form the major contribution of this thesis.
Section 5 compares the works of others on data structure models and top-down design with the products of this thesis. Towards this end, a classification of four different approaches to data structure modeling is created and a common example is expressed in terms of each data structure model. Section 6 presents three arguments for the utility or usefulness of the data structure model and top-down design method. First, it is shown that the data structure model is "complete" in the sense that it covers those data structures charted in Appendix A. The enumeration of each individual data structure and its variations appears in Appendix B. Second, a data base for a large software system is designed using the top-down method. Section 6.2 discusses representative parts of this design; the entire data base is presented in Appendix C. Third, the data structure model's ease of use is compared to the other models categorized in Section 5. Finally, Section 7 discusses further work which arises as extensions of this thesis and summarizes the contributions of this work.

Appendix D is a glossary of terms of significant importance; the definitions describe the terms as they are used in this thesis. Within the body of the text, terms defined in the glossary are underlined when they are first discussed.

1.3 Example Data Structures

The following sections of this thesis contain numerous examples of all kinds of data structures. With a few exceptions, none of these have been created especially for this thesis. Instead, existing published examples and real world data structures from functioning software systems are used. The three major example data structures are:

1. The "organization" data base which describes organizations, their people, and jobs,
2. The "decision table" data structure - an ad hoc implementation of a special decision table algorithm in an assembler, and

3. The "scheduling" data base which records information used to schedule people and machines to accomplish certain jobs.

These example data structures are each described in some detail here.

The major features of the organization data base are shown in Figure 1-1; this data structure contains information about the various organizations which make up a company, and the people and jobs belonging to the organizations. This data base is an extended and modified revision of a data base presented in [CODASYL 1971a, pp. 206-217]. It has been modified to better demonstrate more complex data structures and to contain an illustration of each of the data structures to be surveyed in Section 2.

Figures 1-2 through 1-4 define and describe each of the individual data elements from Figure 1-1. A few aspects of these descriptions deserve further mention at this time. First, ORGANIZATIONS within the company are related in the typical hierarchical manner using the REPORTO and SUBORG data elements. The association connectors shown in Figure 1-1 indicate that REPORTO provides an association between two ORGANIZATIONS and that the association is on a one-to-one basis, i.e., each ORGANIZATION reports to exactly one other ORGANIZATION. Similarly, the SUBORG data element indicates a one-to-n association between an ORGANIZATION and its subordinates.
FIG. 1-1 ORGANIZATION DATA BASE.
<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>ENGLISH NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORGANIZATION</td>
<td>ORGANIZATION DATA BASE</td>
<td>ALL THE INFORMATION ABOUT A COLLECTION OF HIERARCHICALLY RELATED ORGANIZATIONS</td>
</tr>
<tr>
<td>ORGCODE</td>
<td>ORGANIZATION CODE NUMBER</td>
<td>IDENTIFYING NUMBER OF AN ORGANIZATION</td>
</tr>
<tr>
<td>DIVCODE</td>
<td>DIVISION CODE NUMBER</td>
<td>ORGCODE CONSISTS OF TWO PARTS, THE DIVCODE,...</td>
</tr>
<tr>
<td>DEPTCODE</td>
<td>DEPARTMENT CODE NUMBER</td>
<td>... AND THE DEPTCODE</td>
</tr>
<tr>
<td>ORGNAME</td>
<td>ORGANIZATION DESCRIPTIVE NAME</td>
<td></td>
</tr>
<tr>
<td>JOB</td>
<td></td>
<td>JOBS AND PROJECTS EITHER CURRENTLY ASSIGNED TO THE ORGANIZATION, OR PENDING APPROVAL</td>
</tr>
<tr>
<td>JOBCODE</td>
<td></td>
<td>IDENTIFYING NUMBER OF JOB OR PROJECT</td>
</tr>
<tr>
<td>AUTHQUAN</td>
<td>AUTHORIZED QUANTITY</td>
<td>MANPOWER AUTHORIZED FOR JOB CURRENTLY ASSIGNED TO ORGANIZATION</td>
</tr>
<tr>
<td>AUTHSAL</td>
<td>AUTHORIZED SALARY</td>
<td>SALARY EXPENDITURE FOR JOB CURRENTLY ASSIGNED TO ORGANIZATION</td>
</tr>
<tr>
<td>APPSTAT</td>
<td>APPROVAL STATUS</td>
<td>STATUS OF TENTATIVE JOB FOR WHICH APPROVAL IS BEING SOUGHT</td>
</tr>
</tbody>
</table>

**FIG. 1-2. ELEMENTS OF ORGANIZATION DATA BASE, PART 1.**
<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>ENGLISH NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPORTO</td>
<td>REPORT TO</td>
<td>THE SINGLE ORGANIZATION WHICH THIS ORGANIZATION REPORTS TO</td>
</tr>
<tr>
<td>SUBORG</td>
<td>SUBORDINATE ORGANIZATIONS</td>
<td>ZERO, ONE, OR MORE ORGANIZATIONS WHICH REPORT TO THIS ORGANIZATION</td>
</tr>
<tr>
<td>BUDGET</td>
<td></td>
<td>CONSISTS OF...</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>...SALARY FOR EACH PERSON ASSIGNED TO THE ORGANIZATION...</td>
</tr>
<tr>
<td>EMPNO</td>
<td>EMPLOYEE NUMBER</td>
<td>MATCHES ID</td>
</tr>
<tr>
<td>EMPSAL</td>
<td>EMPLOYEE SALARY</td>
<td>... AND A TOTAL AMOUNT FOR SUPPLIES</td>
</tr>
<tr>
<td>SUPPLIES</td>
<td></td>
<td>INFORMATION ABOUT EACH EMPLOYEE CURRENTLY ASSIGNED TO THE ORGANIZATION</td>
</tr>
<tr>
<td>PERSON</td>
<td></td>
<td>SAME AS EMPNO</td>
</tr>
<tr>
<td>ID</td>
<td>IDENTIFICATION NUMBER</td>
<td>CONSISTS OF...</td>
</tr>
<tr>
<td>EMPNAME</td>
<td>EMPLOYEE NAME</td>
<td>...YEAR,</td>
</tr>
<tr>
<td>BIRTH</td>
<td>BIRTHDATE</td>
<td>...MONTH,</td>
</tr>
<tr>
<td>YEAR</td>
<td></td>
<td>...AND DAY OF BIRTH</td>
</tr>
<tr>
<td>MONTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIG. 1-3. ELEMENTS OF ORGANIZATION DATA BASE, PART 2.**

-9-
<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>ENGLISH NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSAL</td>
<td>CURRENT SALARY</td>
<td>PREVIOUS 5 SALARIES IN CHRONOLOGICAL ORDER</td>
</tr>
<tr>
<td>SALHIST</td>
<td>SALARY HISTORY</td>
<td>EMPLOYEE'S EXPERIENCE AS A COLLECTION OF SKILLS</td>
</tr>
<tr>
<td>SKILL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKILCODE</td>
<td>SKILL CODE</td>
<td>IDENTIFYING NUMBER FOR A SKILL</td>
</tr>
<tr>
<td>SKLYRS</td>
<td>SKILL YEARS</td>
<td>NUMBER OF YEARS EXPERIENCE IN THE CORRESPONDING SKILCODE</td>
</tr>
<tr>
<td>CURJOB</td>
<td>CURRENT JOB</td>
<td>JOB ASSIGNMENT; ANY NUMBER OF PEOPLE ARE ASSIGNED TO A SINGLE JOBCODE</td>
</tr>
<tr>
<td>ACTIVE</td>
<td></td>
<td>EMPLOYEE STATUS; ALWAYS ONE FOR ACTIVE EMPLOYEE</td>
</tr>
</tbody>
</table>

FIG. 1-4. ELEMENTS OF ORGANIZATION DATA BASE, PART 3.
Next, Figure 1-1 also shows an important distinction between data elements which occur only once and those which can occur any number of times. This difference is shown by enclosing those elements for which multiple occurrences are possible in a double-sided box. For example, an ORGANIZATION has many PERSONs but just one BUDGET. Further comments on the importance of this sometimes overlooked distinction will be presented in Section 2.

Finally, an ORGANIZATION may be concerned with two kinds of JOBS: jobs currently assigned to it and jobs for which approval is being sought. In these two cases, the data structure has the two different formats shown in Figure 1-1. The description of a single JOB may have either two or three components; a single ORGANIZATION may have jobs of both kinds.

The organization data base will be used exclusively in Section 2 as a source of examples for each of the data structures identified in Appendix A. Figures 1-2 through 1-4 provide documentation for many of those examples.

The decision table data structure was developed as part of an assembler for a specialized programming language. The assembler has been described in [Barton 1970] but usage of the decision table was not covered. The decision table is used to select the appropriate machine instruction from a list of alternatives according to current values of a number of conditions. The conditions represent the options which the programmer has specified in the symbolic version of the instruction.

Figure 1-5 illustrates a sample decision table as it is conceived by the system programmer responsible for the assembler. The decision tables are actually present in the source code of the assembler in approximately this form.
<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 REGISTER, RELOCATABLE ADDRESS, NO INDEX REGISTER, NO C OPTION</td>
<td>GENERATE LOAD FORM 1</td>
</tr>
<tr>
<td>1 REGISTER, ABSOLUTE ADDRESS, INDEX REGISTER</td>
<td>GENERATE LOAD FORM 2</td>
</tr>
<tr>
<td>RELOCATABLE ADDRESS, INDEX REGISTER</td>
<td>GENERATE LOAD FORM 3</td>
</tr>
<tr>
<td>2 REGISTERS, C OPTION</td>
<td>ERROR TYPE 703</td>
</tr>
<tr>
<td>2 REGISTERS</td>
<td>GENERATE LOAD FORM 4</td>
</tr>
</tbody>
</table>

FIG. 1-5. DECISION TABLE FOR LOAD INSTRUCTION.
Each row of the decision table indicates an action to be performed if all the specified conditions are satisfied. The order of the rows is significant. The top row is always the first choice; only if it is not satisfied will the second row be considered. If the second row's conditions are not all satisfied, then the third row will be checked, and so on. In effect, each row includes the implicit condition that all previous (higher) rows are not satisfied. (This ordering restriction allows for considerable abbreviation, especially in large decision tables.) For instance, the third row in the decision table of Figure 1-5 indicates that a "LOAD form 3" will be generated whenever the two specified conditions are true and it is impossible to select either "LOAD form 2" or "LOAD form 1".

This ad hoc structure is of interest because of its implementation using the "coded condition mask" technique for processing decision tables [Oerter 1968], which is a variant of the "rule mask" technique [Kirk 1965]. To explain the resulting data structure and its processing technique, the decision table of Figure 1-5 is transposed into a more traditional form in Figure 1-6. This modified decision table contains exactly the same information as the original, including the preference for what is now the leftmost action. However, the particular implementation in the assembler made a further restriction (for efficiency of storage). The decision tables may contain only conditions which must be true. In effect, each condition in a row of Figure 1-5 becomes a "Y" in Figure 1-7 and all other squares are filled with "\*". This requires some duplication; some conditions and their complements must appear in the decision table. For instance, in the original decision table in Figure 1-5, both "no index register" and "index register" appear. Assuming these two are independent conditions (which, of course, they are not), Figure 1-7 really represents the traditional
<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>GENERATE LOAD FORM 1</th>
<th>GENERATE LOAD FORM 2</th>
<th>GENERATE LOAD FORM 3</th>
<th>ERROR TYPE 703</th>
<th>GENERATE LOAD FORM 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 REGISTER</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2 REGISTERS</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>RELOCATABLE ADDRESS</td>
<td>Y</td>
<td>*</td>
<td>Y</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ABSOLUTE ADDRESS</td>
<td>*</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>INDEX REGISTER</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C OPTION</td>
<td>N</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Y</td>
</tr>
</tbody>
</table>

Y - YES  
N - NO  
* - DON'T CARE  

FIG. 1-6 TRADITIONAL DECISION TABLE.
<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GENERATE LOAD FORM 1</td>
</tr>
<tr>
<td>1 REGISTER</td>
<td>Y</td>
</tr>
<tr>
<td>2 REGISTERS</td>
<td>*</td>
</tr>
<tr>
<td>RELOCATABLE ADDRESS</td>
<td>Y</td>
</tr>
<tr>
<td>ABSOLUTE ADDRESS</td>
<td>*</td>
</tr>
<tr>
<td>INDEX REGISTER</td>
<td>*</td>
</tr>
<tr>
<td>NO INDEX REGISTER</td>
<td>Y</td>
</tr>
<tr>
<td>C OPTION</td>
<td>*</td>
</tr>
<tr>
<td>NO C OPTION</td>
<td>Y</td>
</tr>
</tbody>
</table>

Y YES
* DON'T CARE

FIG. 1-7. MODIFIED TRADITIONAL DECISION TABLE.
decision table version of Figure 1-6. Because of this restriction, the algorithm and resulting data structure vary slightly from that described in [Oerter 1968].

Now, the decision table of Figure 1-5 can be implemented as follows, using the (modified) coded condition mask technique. The decision table is stored in memory as a list of conditions and a "false vector" for each condition. Each false vector corresponds to a single row of a decision table as shown in Figure 1-7, and is formed by encoding Y as 0 and * as 1. The false vectors for Figure 1-7 are the following:

1 register: 00111
2 registers: 11100
relocatable address: 01011
absolute address: 10111
index register: 10011
no index register: 01111
C option: 11101
no C option: 01111

Clearly this encoding results in no loss of information; it is also the basis of a simple algorithm for executing the decision table. In the algorithm, the false vector is used to eliminate possible actions when the corresponding condition is not satisfied.

Figure 1-8 shows an example execution of the decision table assuming the listed values for the conditions. Execution begins with an "initial truth vector" of all 1s. Then each condition is examined. If a condition is true, nothing is done. If a condition is false, the initial truth vector is ANDed with the condition's false vector (hence the name) and the result replaces the initial truth vector. After all conditions in the decision table have been processed, the result vector contains a 1 for each possible action. The leftmost 1 corresponds to the preferred action. If the 1 is in the nth position from the left, the nth action is selected.
FIG. 1-8. EXECUTION OF DECISION TABLE.
In Figure 1-8, since four conditions are FALSE, four ANDs are performed. The result vector contains more than one 1, but because the rows of the original decision table (Figure 1-5) were ordered, the leftmost 1 is the proper choice.

This brief introduction to the coded condition mask algorithm is necessary to understand the examples based on the decision table which are used in this thesis. The top-down design of the entire data structure (Section 4.3.1) is heavily influenced by this algorithm.

The scheduling data base was first presented in [Frank and Sibley 1973] where it was used to illustrate the features of the CODASYL Data Base Task Group proposal [CODASYL 1971]. Since then the same example data base has also been expressed in terms of a relational model by [Codd and Date 1974].

This data base is pictured in Figure 1-9; its main purpose is to interrelate people, machines, and scheduling information. Some other information of a personnel department nature is also kept for each person. Using this data base, people can be scheduled to work on machines. The data base records various skills or abilities which each person has and also, for each machine, a list of skills, any one of which qualifies a person to operate the machine.

Figure 1-9 shows the data base approximately as drawn in [Frank and Sibley 1973, p. 8]; the drawing has been modified to use the box-within-a-box notation for multiple occurrences of a data element, similar to Figure 1-1. Thus, for example, each MACHINE includes some number of SCHED INFOS, each one detailing which person will be working on the machine for some specific future time period. The arrows correspond to the associations of Figure 1-1. The numbers on the ends of
FIG. 1-9. SCHEDULING DATA BASE.
the arrows show how many of each of the elements are connected together. For example, each PERSON is attached to \( N \) different JOBS which he/she held some time in the past.

The specific kinds of information to be stored and the names given them in Figure 1-9 are:

1. Information about individual employees (PERSON),
2. Medical or absence information about employees (MEDICAL),
3. Prior job history for each person (JOB),
4. Education information for each person (EDUC INFO),
5. Information about individual machines used in the manufacturing process (MACHINE),
6. Scheduling information (SCHED INFO), i.e., which people will be working on which machines for some future period, and
7. Skill information (SKILL), i.e., which skills are possessed by which people and required to operate which machines.

The individual components of each of these kinds of information (e.g., that PERSON consists of name, age, sex, etc.) are straightforward and will not be detailed here. Instead, the components will be introduced as needed throughout the examples based upon the scheduling data base.

These three examples include a wide variety of information requirements. Each example will be used repeatedly
throughout the thesis. In most cases just one part of the
data bases will be used for a single example. For instance,
examples based on just the JALHIST element of the organiza-
tion data base and upon the PERSON-SKILL-MACHINE portion of
the scheduling data base are used. However, almost every
aspect of each of the examples is covered in detail some-
where. A complete top-down design of the decision table is
carried out in Section 4.3.1 and a similar design for the
entire scheduling data base is presented in Section 4.4.
Individual elements and various subsets of the organization
data base are used in the following section to describe each
of the common data structuring techniques.
2. **COMMONLY USED DATA STRUCTURES**

Data structures are a fundamental part of computer science, as such, have been investigated for some time. This section examines the data structures commonly provided by current programming languages and data base management systems; their purpose is to:

1. Present the contemporary mélange of data structures supplied for programmers and data base designers,

2. Show the similarity and overlap between programming languages and data base management systems with regard to data organization and structure, and

3. Provide a concise comparison between data structures in 21 different systems (languages and data base management systems).

This background will be appealed to in Section 3 of this thesis where a concise model for the collection of common data structures is introduced. The various data structures are also used for numerous examples and to illustrate special techniques throughout the thesis. Section 6.1 demonstrates that the data structure model is "complete" in the sense that it provides all the data structures surveyed here.

The major portion of what follows in Section 2 is a description of each individual data structure tabulated in Appendix A. A general definition and an example from either a programming language or data base management system are presented for each data structure. Prior to this enumeration of data structures, some general trends in data description are noted.
The various sorts of data structures are grouped into four classes and presented in tabular form along the left side of the charts in Appendix A. The data structures provided by each of the 21 programming languages and data base management systems are noted in this table. (References to descriptions of each system are also supplied at the top of the charts in Appendix A). A word of caution concerning use of this table to compare systems: Appendix A is a very incomplete comparison of the 21 systems since it considers only data structures. Many other aspects in addition to data structuring features must be considered for a complete comparison of either data base management systems or programming languages. More complete comparisons are provided by some of the references mentioned below. In Section 2.3, some observations and conclusions which do seem appropriate are drawn from Appendix A.

Truly generalized data base management systems have become increasingly popular in recent years; these systems provide a wide variety of data organizations. The current controversy between the pointer-based or network model and the relational model for data base design (see [Codd 1971; Bachman 1973; Codd and Date 1974; Date and Codd 1974]) has uncovered several basic questions about storing information. The network model is represented here by the proposal of the CODASYL Data Base Task Group (abbreviated CODASYL DBTG from now on), IDS, IMS, and TDMS. ALPHA, LEAP, and MacAIMS are all based on versions of the relational model.

In the area of programming languages, new languages have introduced advanced data structures motivated by mathematics and programming experience. The new languages with the most interesting data structures and some traditional, popular languages are included in Appendix A. Considering this wealth of data structures, it seems very timely to recount them all in one place.
Some previous surveys of data structures have appeared; however, none have concentrated, as does the following, exclusively on the data structures actually furnished to the users of programming languages and data base management systems. The most interesting surveys are [Dodd 1969; Williams 1971; Gray 1967; D'Imperio 1969; Hoare 1972]. [Dodd 1969] discusses some of the data structures listed in the aggregates and associations sections of Appendix A with emphasis on file concepts for data bases. [Williams 1971] and [Gray 1967] both survey the rich variety of data structures which have proven their usefulness to various computer graphics applications. [D'Imperio 1969] distinguishes (logical) data structure from storage structure while presenting the data structures in a rather esoteric collection of programming languages. The most encompassing view of data structures and their use in programming languages is the conference proceedings [Tou and Wegner 1971]. [Hoare 1972] discusses a limited collection of data structuring techniques useful for the abstract design of data structures and describes methods for manipulating and implementing them.

Numerous reviews, surveys, and tutorials on data base management systems have appeared. The best known and most detailed survey is [CODASYL 1971a] which covers 10 systems including some of those included in Appendix A (CODASYL DBTG, IDS, IMS, and TDMS). Up-to-date introductions to the current state of the field are provided by [Engles 1972], [Cagan 1973], and [Everest 1974]; the latter considers current problems from data integrity to legality. [Lyon 1971] is a tutorial on the motivation and implementation of various file structures for data base management systems. Data management techniques for file organisation are covered by [McGee 1969] and [Senko 1969], the latter considering
text retrieval and other specialized data base management systems. [Bachman 1972] traces the evolution which lead to current network file structures and presents 11 basic concepts for structural storage elements. The other side of the network vs. relational controversy mentioned above is covered in tutorials [Date 1974] and [Whitney 1973] both of which motivate the relational model through comparison with network schemes. Data base management has been the featured cover subject on two issues of DATAMATION (October, 1972, and September, 1974); both provide popularized introductions to the field. A continuing view is provided by the publications of the ACM special interest group SIGMOD (formerly SIGFIDET).

2.1 General Trends in Data Description

Some basic trends have evolved in the field of data base management systems. These trends flavor the view of data structures presented below; the trends are:

1. Explicit data definition,
2. Separate data definition dictionary, and
3. Data independence.

Early data base management systems (e.g., IDS) introduced the idea of describing the organization, format, and structure of their data bases with special declarations. These declarations define the form for each component part of the data base; when in use the actual data base contains numerous pieces of information each of which are formed according to the declarations. This explicit data definition exists independently of any piece of information formed according to its declarations. Thus, it is reasonable to
think of the data definition on one hand, and any number of "occurrences" or "instances" of the definition on the other. It is the collection of current instances which form the actual information content of the data base at any given moment.

A data definition is also sometimes called a "schema" (as popularized by various CODASYL reports, including [CODASYL 1971] and [CODASYL 1971a].) A dictionary definition of the term is enlightening.

schema n. a diagram, plan, or scheme.

Thus a schema or data definition is a diagram or plan for all the instances in the data base.

In the discussion of data structures which follows, the terms data definition and instance will be used exclusively. Numerous figures will be used to present the various data structures. These figures illustrate a sample data structure (drawn from the organization data base discussed in Section 1.3) by presenting a data definition on the left side of the figure and one or more instances on the right. The data definition is drawn in a block diagram form which should be applicable to all programming languages or data base management systems. Each figure also includes a sample declaration of the data structure in some system. This collection of figures provides a complete description of all the data structures presented in Appendix A.

Not all systems distinguish between data definition and instance. The alternative is to let a data structure and one instance be defined simultaneously; in effect every instance has its own, exclusive data definition. The only
way to define two instances with the same structure is to duplicate the data definition. This approach makes it hard to tell when two instances are exactly the same. Appendix A, in its bottom section, notes which systems distinguish between data definition and instance.

Once explicit data definitions are provided, the next step is to gather all the definitions together and create a complete "dictionary" of the data base's structure. This separate dictionary can be accessed as needed by a language compiler or run-time input/output subroutines in order to properly access the data. Once this step is taken, the data definition need not be based upon any single language. Thus, the way is opened for sharing of definitions between programs written in several languages. The CODASYL DBTG "schema" is a language-independent dictionary of explicit data definitions.

The primary reason for creating a separate dictionary of data definitions is to grant "data independence" to the programs using the data base. Data independence is well defined in [Engles 1972, p. 52] as follows:

"Data independence is the capability by which an application program is insulated from the various aspects of data bank design and implementation. A high degree of data independence implies the ability to make changes to a data bank, such as a change to the method of representing a complex data map, without requiring changes to source programs."

Thus, data independence is intended to make it easier to write programs which will continue to operate despite changes in their data organization. Of course at some point, after extensive modification of the data, the program can no longer operate properly even with the aid of the most elaborate data independence scheme. There has been much discussion
of just how much data independence systems should attempt to provide. A much stricter and perhaps more attainable definition of data independence, given in [Date and Codd 1974, p. 31], is:

"(a) program immunity to change in the storage structure (sometimes referred to as 'physical data independence'); also

(b) program immunity to growth in the data model definition (sometimes referred to as 'logical data independence')."

Note that change in the conceptual or logical data definition is not mentioned. Instead, only growth is provided; i.e., new data may be added without affecting the current programs (which do not use the new data).

Systems which provide various degrees of data independence often do so with the aid of a "subschema" in addition to the schema ([CODASYL 1971; Date and Hopewell 1971; Date and Hopewell 1971a]). The subschema is a separate explicit data definition specialized for an individual program. It records just the information which a program needs to do its job; thus, it is basically a subset of the schema although some differences (e.g., alternative names) may be allowed. IMS uses its "sensitivity" concept in a way similar to a subschema (see the reference given in Appendix A).

Another of the appeals of data independence is that it aids, or at least provides a starting point for, some approaches to data integrity and data protection. These two related and desirable concepts are just beginning to be attained by data base management systems. Data integrity means keeping the data structure well formed, e.g., preventing
linked structures from becoming unconnected. Data integrity also concerns the consistency of the information represented by the data structure, e.g., making sure that if a count word says there are five records in a file then there are indeed five records there. Data protection usually means restricting access to information to prevent unauthorized changes or for reasons of privacy. [Browne 1971] is a good overview of these topics and introduces a conference session covering privacy and data integrity.

Figure 2-1 depicts a generalized block diagram of a data base management system in which data independence is provided. The user makes data base accesses based either upon a subschema or knowledge of the data definition recorded in the dictionary. The data base management system accesses the actual data base from some physical storage medium, again using the stored data definitions.

2.2 Common Data Structures

This section is a catalog of the data structures tabulated in Appendix A. Each data structure is described with the aid of an example drawn from the organization data base. For each data structure a pictorial data definition and one or two possible instances are shown. Additionally, a data declaration from one of the 21 systems used in Appendix A is shown for the example data structure. In the example declarations from the programming languages and data base management systems the actual keywords in the language are shown underlined while the terms supplied by the user are in normal type. Whenever possible, the complete declaration is shown; however, three dots may sometimes be used to represent sections of the declaration which are not essential for understanding the example. The name of each data structure from Appendix A is underlined when it is first described; however, these names do not appear in the glossary (Appendix D).
FIG. 2-1. DATA INDEPENDENCE DBMS.
A few primitive or atomic terms form the basis for much of the following and will be discussed first. The examples will often speak of a "collection" of data "elements" of some kind or another. For instance, a collection of information about people would consist of one element for each person.

Finally, another word of caution in regard to Appendix A: this table records just what the systems directly provide. Data structures which may be implemented in terms of data structures in the systems are not entered in the table. Appendix A summarizes just those data organizations which the systems allow the user to explicitly define. An example may make this distinction clearer. Figure 2-2 shows in its first line the definition of a typical one-dimensional array in the programming language ELI. The single dimension array is a concept which the language explicitly makes available. ELI does not, however, have a corresponding declaration for matrices of dimension two or greater. Such a data structure can be easily implemented in ELI, as shown in the second line of Figure 2-2, by nesting declarations. In this example, a two-dimensional matrix is implemented as an array of arrays. In Appendix A, ELI is noted as providing array data structures but not matrix data structures since the language explicitly provides the former but not the latter. This distinction between which data structures a system explicitly provides and those which can be built-up from them is maintained throughout Appendix A.

The data structures will be discussed in four classes, as they are grouped along the left side of the charts in Appendix A. These four classes, basic items, aggregates, associations, and files, were adapted from [CODASYL 1971a] where six generic structure classes were defined in the process of comparing data base management systems. The approximate correspondence between the terms used here and in
FIG. 2-2. EL1 ARRAY AND MATRIX.
[CODASYL 1971a] is shown in Figure 2-3. The two classes not used here are of interest only when access of data bases is considered. Within the left-hand headings of Appendix A, alternative names for some of the classes and individual data structures are shown in parentheses. The upper, unparenthesized term is used in the following discussions. A description of each of the four classes of data structures begins each main subsection below.

2.2.1 Basic Items

Basic items are the primitive elements from which data structures are built. However, they exhibit enough diversity to be considered data structures in their own right. Some basic items, such as numbers and strings are quite simple; others, such as the virtual conceptual items shown in Appendix A, have much to offer the sophisticated data base designer. Nevertheless, all basic items have something in common: they are indivisible.

2.2.1.1 Storage

Some systems allow the user to play intimately with the very protyle of data structures: raw storage. Data structures designed in this way are usually inseparable from the memory structure of the host computer; the distinction between data definition and instance is rarely made and data independence vanishes. The power of such a scheme is undeniable - the user can develop any data organization he/she desires. However, with the rich collection of data structures detailed below, there would seem to be little need to invent new ones.

No attempt will be made here to create a data definition or schema for storage. Instead a possible use of storage for
<table>
<thead>
<tr>
<th>CODASYL FEATURE</th>
<th>TERM USED HERE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYSIS</td>
<td></td>
</tr>
<tr>
<td>TERM</td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td>BASIC ITEM</td>
</tr>
<tr>
<td>GROUP</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>GROUP RELATION</td>
<td>ASSOCIATION</td>
</tr>
<tr>
<td>ENTRY</td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td>FILE</td>
</tr>
<tr>
<td>DATA BASE</td>
<td></td>
</tr>
</tbody>
</table>

*FIG. 2-3. COMPARISON OF TERMS.*
the organization data base is shown in Figure 2-4. The first line of the example data declaration (using IBM 360/370 assembly language) allocates the next full word of storage and associates the name ORGCODE. (In most computers, memory occurs in some standard "full word" amount which is most conveniently accessed by the instruction set.) Likewise, most assembly languages allow storage to be allocated in terms of bits within a full word and multiples of full words. The next line allocates a single bit from the next storage location and associates the name SEX with the storage. The final line allocates five full words for ORGNAME.

2.2.1.2 Pointer

Some think of pointers as the shibboleth of the experienced programmer. A pointer is a name, address, or reference to some other element; they evolved from assembly language programming where they are useful for creating complex data structures from raw storage. The preeminent tutorial on definition and use of data structures in this manner is [Knuth 1969, chp. 2].

Pointers also exist in numerous high level programming languages in two basic forms: general address and qualified by type. The first kind of pointer, the general address, may be thought of as an indicator which can refer to any arbitrary element. An example data definition and use of this kind of pointer is shown in Figure 2-5. The general address pointer is most similar to the way addresses are used in assembly languages. As shown in Figure 2-5 one instance of PTR1 could point to a number while another points to a character string. Even the same instance of a general address pointer can refer to two kinds of elements at two different times. Such is the case with the sample BLISS statements also shown in Figure 2-5.
FULL WORD: ORCODE DS 1F

PARTIAL WORD: SEX DS BL.1

MULTIPLE WORDS: ORGNAME DS 5F

FIG. 2-4. STORAGE.
FIG. 2-5. GENERAL ADDRESS POINTER.

OWN PTR1;
PTR1 ← ORGCODE; !PTR1 CONTAINS ADDRESS OF ORGCODE
...
PTR1 ← ORGNAME;
The other kind of pointer is restricted to refer only to elements of certain types (for more on types, see Section 2.2.1.3). This kind of pointer is more appropriate to "typed" high level languages (as opposed to "typeless" ones, again see Section 2.2.1.3). A type qualified pointer is shown in Figure 2-6; this pointer is restricted to point only to elements of type ORGANIZATION (which would be declared elsewhere). The pointer REPORTO could never point to a SKILL or an ORGCODE.

The difference between general address and type qualified pointers is quite significant. Type qualified pointers can be an aid. The restriction that they point to only a certain kind of object is enforced by the programming language compiler, and the proponents of this kind of pointer claim this yields more reliable programs. Type qualified pointers can also be a hinderence when the programming language is not well suited to the program being written. In this case, a general address pointer allows the programmer to do whatever he/she desires without interference from the compiler.

Storage and general address pointers comprised the programmer's basic data structure building blocks prior to the introduction of high level languages with more sophisticated data structures. These building blocks were used to create from scratch many of the data structures described in the rest of this section. It was this experience in using these data structures which lead to their inclusion in programming languages.

2.2.1.3 Conceptual Items

Instead of providing just raw storage, most systems furnish data elements to represent real world objects such
FIG. 2-6. TYPE QUALIFIED POINTER.

ALGOL 68

REF ORGANIZATION REPORTO;

5331 SOFTWARE R & D....

5325 SOFTWARE PRODUCTION....
as numbers and characters. These different sorts or kinds of data elements are termed "types". Example types are "integer" and "organization"; corresponding data elements are thought of as being of "type integer" or "type organization".

A distinction is made between "typed" and "typeless" languages. Typed languages distinguish between the data elements by associating a type with each one. Typeless languages treat all data elements identically. Figure 2-7 shows some declarations of conceptual items in the typeless language BLISS and in typed FORTRAN. In BLISS, all the data elements are defined the same way; thus, the compiler cannot distinguish between them. In FORTRAN, the declarations associate the types "logical", "real", and "integer" with the data elements. In this case the compiler can detect and possibly correct (by automatic "type conversion") cases where a data element is used in an improper context.

Consider the two assignment statements:

\[
\begin{align*}
AGE & \to AGE + 1 \\
ORGNAME & \to ORGNAME + 1
\end{align*}
\]

A typeless language assumes the programmer's instructions are appropriate and would allow both assignments. A typed language would object to the second assignment since addition to type "character string" is meaningless. Of course, if the programmer knew ORGNAME was really a number in certain cases, a typeless language would let him/her take advantage of the special case. Also, with a typeless language the compiler will never get in the programmer's way by converting between types behind the programmer's back and possibly to the program's detriment.
FIG. 2-7. TYPELESS AND TYPED LANGUAGE DECLARATIONS.
Given the distinction between typed and type-less languages, a further refinement is useful among typed languages. Some languages allow types to be selected only from a built-in set of types. FORTRAN, for example, provides the three types used in Figure 2-7 plus a "complex" type and no others. Other systems start out with a set of built-in types and allow the user to define new types. The new types are built up from the language supplied ones using special type construction methods. An example construction of a user defined type is shown in Figure 2-8; the type defined is a hierarchy structure suitable for representing some of the information in an ORGANIZATION data element. Type construction methods from many programming languages and data base management systems are used in various examples from here on. These methods give the systems the ability to represent many of the aggregate data structures discussed in Section 2.2.2. Programming languages which allow the user to define both new types and operations on the types are called "extensible" languages (for more information see the conference proceedings [Schuman 1971]). Appendix A tabulates the type-related distinctions between various programming languages.

One more aspect of typed languages which allow user defined types deserves mention now to avoid confusion in later examples. Most of these languages allow two alternative ways of defining a new type. The two ways are equivalent and are illustrated in Figure 2-9. The distinction is whether the new type is given a name or is "spelled out" each time it is used. In the top of Figure 2-9, the new types "sex" and "person file" are really just shorthand so that the full definitions need not be written out in the declarations of each instance of the type. The shorthand is not used in the equivalent declarations presented in the bottom of Figure 2-9.
ALGOL 68

MODE ORGANIZATION =
STRUCT(INT ORGCODE, [1:0 FLEX] CHAR ORGNAME,
REF ORGANIZATION REPORTO);

FIG. 2-8. TYPE CONSTRUCTION.
PASCAL

SEX = (MALE, FEMALE)

PERSONFILE = FILE OF PERSON

THISSEX: SEX

THATSEX: SEX

NEXTFILE: PERSONFILE

---

PASCAL

THISSEX: (MALE, FEMALE)

THATSEX: (MALE, FEMALE)

NEXTFILE: FILE OF PERSON

---

FIG. 2-9. USER DEFINED TYPES WITH AND WITHOUT NAMES.

- 44 -
One of the major uses of types is to define various encodings for conceptual basic items. These encodings are listed in Appendix A. The most familiar encodings are used for numbers; some definitions and instances of regular numbers are shown in Figure 2-10.

A less well known encoding is the tabular data element as depicted in Figure 2-11. It is used when the data element may contain one of a fixed number of possible values. Tabular data elements are exemplified by PASCAL as portrayed in Figure 2-11. (In PASCAL, an order is implied among the possible values - this feature is an inessential aspect of tabular structures in some languages.) Although it represents a common programming practice, tabular data elements are provided only by a few of the systems in Appendix A.

Boolean data elements are, in effect, a special case of tabular ones with just two possible values. The two values are interpreted as true and false; however, a Boolean data element is often used to imitate a two-valued tabular as in Figure 2-12 where the SEX data element is encoded as true or false. Boolean data elements are most often used as program control flags.

A picture data element is defined by an encoded pattern or picture of what the data's format will be. Figure 2-13 uses pictures to define two fields from the organization data base; the pattern scheme used is that of the CODASYL DBTG (9 representing any numeral, V a decimal point). Various pattern schemes exist; they all specify the numerals, characters, and special symbols allowed in each position of the data element. In the example declaration from the CODASYL DBTG, the initial 1 is a hierarchy level number and will be discussed later.
DIVCODE

NAME

OTHER DEFINING INFORMATION

REAL

FIXED POINT

TYPE OF NUMBER

INTEGER

DEFINITIONS

2 DEFINITIONS

1 INSTANCE OF EACH

FIG. 2-10. REGULAR NUMBERS.

DIVCODE

EMPSAL

53

950.25
SEX: (MALE, FEMALE)
DEPTCODE: 1...99

TDMS
1  SEX (NAME) VALUES ARE MALE, FEMALE
1  DEPTCODE (NUMBER) VALUES ARE 1...99

FIG. 2-11. TABULAR.
FIG. 2-12. BOOLEAN.
CODSYL DBTG

1 DIVCODE PICTURE IS "99."

1 EMPSAL PICTURE IS "9999V99."

FIG. 2-13. PICTURE.
Constant data elements always have a single special value. They are used to make data structures more readable, to add redundancy for error checking, or for future compatibility with other data structures. In Figure 2-14, it is assumed that the ACTIVE data element is one for active employees (as opposed to retired, laid off, etc.). If the organization data base contains data only for active employees, ACTIVE will be a constant as shown.

Finally, special data elements are used in many systems for miscellaneous kinds of information. For instance, some provide date, time, or weight elements. Others provide system-dependent data elements that store special information useful within the system. Figure 2-15 demonstrates both kinds of special data elements. The CODASYL DBTG "data base key" is a unique identifier of "records"; thus, if REPORTO is such a special element, it can reference any ORGANIZATION record. Special data elements are usually added to a system to avoid the need for an extensible type scheme, to supply special features of the system, or to suit a particular application area. Another use of special data elements is "type variables"; i.e., an element whose values are types (as described earlier in Section 2.2.1.3). The MODE declaration in ELL1 defines type variables.

All the above have been encodings; they all encode various kinds of information into a standard amount of storage, usually a full word as in Section 2.2.1.1. For strings, a reasonable size cannot be assumed; there are two alternatives as shown in Appendix A. To use a fixed length string, the user must declare either the length or the maximum length of the characters to be stored. Figure 2-16 shows a fixed length string for the ORGNAME data element. If any string shorter than 20 characters is assigned to ORGNAME, most systems will provide automatic padding to
CONSTANT

ACTIVE

1

CONSTANT VALUE

1

ACTIVE

1 INSTANCE

DEFINITION

ALGOL 68

INT ACTIVE = 1

FIG. 2-14. CONSTANT.
RIQS

RECORD DEFINITION
(1) BIRTH

DATA RESTRICTIONS
DATE (1)

CODASYL DBTG

1 REPORTO TYPE IS DATABASE - KEY.

FIG. 2-15. SPECIAL DATA.
ALGOL 68

$$[1:20] \text{CHAR ORGNAME;}$$

FIG. 2-16. FIXED LENGTH STRING.
fill the entire amount of space allocated with additional blanks. Variable length strings, an example of which is shown in Figure 2-17, automatically adjust the amount of storage allocated whenever a new value is assigned. Figure 2-17 exhibits a "flexible" string in Algol 68.

Regular numbers and strings are the traditional basic data elements for both programming languages and data base management systems. With only one exception, every system in Appendix A provides some kind of numbers and strings. The one exception is FORTRAN which does not supply strings.

There remains one more kind of conceptual item which is as rare as strings and numbers are common. Virtual data elements are used to define basic items that "are not really there." The items are, instead, stored elsewhere or produced by a function when needed. A functional virtual item is demonstrated in Figure 2-18 using the AGE data element from within the information about a PERSON. A person's age can be calculated from his/her birthdate (assuming that the current date is available for use by the function). A functional AGE item is declared in CODASYL DBTG in Figure 2-18 (which assumes the BIRTH item and the function COMPUTEAGE are defined elsewhere). Since AGE changes regularly, computing it each time it is needed is an ideal method for keeping the data base up-to-date. These advantages of functional virtual items have also been pointed out in [Bobrow 1972] which described "functional data items" as part of an experimental data base system and in [Date and Hopewell 1971] which suggests "computed virtual fields." [Polinus et al. 1974] views the information content of a data base on a continuum from completely physical (i.e., actually present) to entirely virtual.
ALGOL 68

\[ \texttt{[1:0 FLEX] CHAR ORGNAME,} \]

FIG. 2-17. VARIABLE LENGTH STRING.
FUNCTIONAL VIRTUAL ITEM

AGE

COMPUTEAGE (BIRTH)

FUNCTION AND ANY PARAMETERS

AGE

27 = VALUE OF COMPUTEAGE (10/8/47)

DEFINITION

1 INSTANCE

CODASYL DBTG

1 AGE IS VIRTUAL RESULT OF COMPUTEAGE USING BIRTH.

FIG. 2-18. FUNCTIONAL VIRTUAL ITEM.
The second type of virtual item is equally useful. An elsewhere virtual item is kept in some other part of the database. In Figure 2-19, current salary (the CURSAL item) for a PERSON is a virtual item which is really stored with the BUDGET information for the ORGANIZATION. The example declaration, again from CODASYL DBTG, assumes an owner-member structure named SALPERSON between SALARY information and PERSON with ID equal to EMPNO.

Both kinds of virtual items are used to avoid redundancy in the database. In the example in Figure 2-19, a person's salary can be stored in just one place. This also eases update of the database since there is no need to change salary information in two places. The example in Figure 2-18 demonstrates how virtual items can keep a database current as well as eliminate redundancy.

2.2.1.4 Equivalence

One last topic, equivalence, completes the discussion of basic items. Equivalence is an old concept which remains important today. There are two different sorts of equivalence which are quite different. They are described in Appendix A as equivalence of two things at the same time and equivalence of two alternatives.

Equivalence of two things at the same time is used when a data element can usefully be thought of in two different ways. In the ORGANIZATION data base, ORGCODE may be used as the number for an ORGANIZATION or it may be broken down into a DIVCODE and a DEPTCODE. Any single instance of ORGCODE may be used in both these two ways at the same time, since some programs will treat ORGCODE as an indivisible quantity whereas others will be interested in breaking it down into its parts. This sort of dual
FIG 2-19. ELSEWHERE VIRTUAL ITEM.
identity for a data element is useful for a number of reasons. It often adds legibility or changeability to a program, it can be used as a refinement of the data element (like the hierarchy structure discussed below), and it is sometimes used to fool the compiler (perhaps to avoid type checking). All three of these reasons are illustrated by the example shown in Figure 2-20 where a PERSON is refined into different components of different types in FORTRAN.

Equivalence of two alternatives allows a data element to be two (or more) totally different things at different times. The data element is thought of as having several types, but each instance can be only a single type. This kind of data element is most often used as a subroutine input parameter. In order to define general purpose or "generic" subroutines, it is useful to have parameters of different types. A typical example is a parameter which can be either an integer or real number. An example from the organization data base might be an input parameter which is either ID or EMPNAME. Figure 2-21 shows the definition of a data element which indicates a PERSON by either name or number. Such a data element could be an input parameter to a subroutine which prints a report addressed to an employee by either ID or EMPNAME.

2.2.2 Aggregates

Aggregates give structure to a collection of data elements. Appendix A recounts the different kinds of aggregates in two categories, tables and groups. The term aggregate was used in much the same sense by [Sammet 1969, pp. 74-75].
FORTRAN

INTEGER PERSON(6), ID
REAL CURSAL
EQUIVALENCE (PERSON(1), ID), (PERSON(5), CURSAL)

FIG. 2-20. TWO THINGS AT SAME TIME EQUVALENCE.
FIG. 2-21. EQUIVALENCE OF ALTERNATIVES.
2.2.2.1 Table

The various table data structures arose as generalizations from the matrix and array. They all collect data elements together into structures which are often thought of or drawn in a tabular form.

The array structure specifies an ordered collection of basic elements; it was popularized by the earliest high level programming languages. Figure 2-22 exhibits an array for the data element SALHIST which contains an employee's five previous salaries in chronological order. The data elements which compose an array are ordered by an index; in this example the index ranges from one to five representing each of the last five years. Figure 2-22 also introduces a new data definition feature: semantics or documentation may be included in the definition.

Arrays store a collection of data elements under a single name. The different data elements are distinguished by the index. Conceptually, the array of Figure 2-22 is similar to five individual variable names, such as:

SALHIST1
SALHIST2
SALHIST3
SALHIST4
SALHIST5

Programming languages, however, contain the ability to index arrays under program control. Thus, the various individual data elements in an array can be accessed conveniently using the program to calculate the proper index value each time the array is to be accessed. Figure 2-23 shows a sample program segment accessing the SALHIST array. On
HOW ARRAY IS INDEXED

SALHIST

INDEX RANGE: 1-5
IS PREVIOUS YEARS

REAL IS PAST SALARY

TYPE OF ELEMENTS

INDICATES MULTIPLE OCCURRENCES

FORTRAN

DIMENSION SALHIST(5)
REAL SALHIST

FIG. 2-22. ARRAY.
FORTRAN

DO 100 I = 1, 5
   
   SALHIST (I)
   
100 CONTINUE

FIG. 2-23. FORTRAN LOOP ACCESSING ARRAY.
the other hand, most programming languages do not allow construction of variable names at run-time. Thus, a mechanism of the form

\[ \text{SALHIST:} I \]

(where the colon indicates concatenation the current value of the variable I to form a name) is not readily available to access the five single SALHIST\textsubscript{i} variables.

Originally, array indexes varied between one and the maximum number of data elements to be stored. Now, more general indexes have been popularized. The first generalization allowed an index to be any range of consecutive positive or negative integers. The language PASCAL typifies further generalizations which allow indexes to be something other than regular numbers. As shown in Figure 2-24, PASCAL would allow the SALHIST array to be implemented with any tabular basic item (as discussed in Section 2.2.1.3) as the index.

A final generalization of the array structure should be mentioned. Originally, array elements were always basic items and usually regular numbers. However, the programming languages which provide user defined types often allow arrays of arbitrary data structures, for instance, arrays of arrays and arrays of sets.

The array structure is really just a special case of the matrix structures. A matrix may have any number of indexes. The number of indexes is called the "dimension" of the matrix. A matrix with two indexes is displayed in Figure 2-25. Each dimension is defined by giving the range of its index. A single data element is associated with each possible combination of index values. In the example in Figure 2-25, one previous salary is accessed by indicating
PASCAL

SALHIST : ARRAY [PYEARS] OF REAL;

FIG. 2-24. PASCAL SALHIST ARRAY WITH TABULAR INDEX.
**DEFINITION**

<table>
<thead>
<tr>
<th>1000</th>
<th>1000</th>
<th>1000</th>
<th>1000</th>
<th>1000</th>
<th>1025</th>
<th>1025</th>
<th>1025</th>
<th>1025</th>
<th>1025</th>
<th>1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>675</td>
<td>675</td>
<td>675</td>
<td>675</td>
<td>675</td>
<td>750</td>
<td>750</td>
<td>750</td>
<td>750</td>
<td>750</td>
<td>1000</td>
</tr>
<tr>
<td>600</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>650</td>
<td>650</td>
<td>650</td>
<td>650</td>
<td>650</td>
<td>675</td>
<td>675</td>
</tr>
<tr>
<td>533</td>
<td>533</td>
<td>533</td>
<td>533</td>
<td>533</td>
<td>580</td>
<td>580</td>
<td>580</td>
<td>580</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>533</td>
</tr>
</tbody>
</table>

**1 INSTANCE**

ALGOL 68

\[
\begin{array}{cccccccccccc}
  1:5,1:12 & \text{REAL MSALH,}
\end{array}
\]

**FIG. 2-25. MATRIX.**
its month and year. Consideration of programming style may suggest other kinds of index values. Resorting once again to PASCAL, Figure 2-26 shows another version of the monthly salary history.

The matrix structure is often used to devise other structures when they are not provided by the language in use. For example, to eliminate the need for the repeating structure (see Section 2.2.2.2) SKILL within the repeating structure PERSON in the organization data base, a two-dimensional matrix could be used. As shown in Figure 2-27, the matrix could be indexed by the employee number and the skillcode to retrieve the number of years the person held the skill, or 0 if the person had no such experience. As shown, employee number 2 has only skills 1000, 1002 and 1998 for 5, 1, and 2 years respectively. A FORTRAN programmer could implement this rather involved data structure as shown at the bottom of Figure 2-27. The program would have to meticulously convert SKILCODE from the range 1000-1999 to 1-1000 (by subtracting 999 each time). Also, the matrix would probably be sparse (many zero elements) if most employees had only a few skills. A proposal to recognize sparse arrays as a true data structure and some suggested implementations are provided by [Hoare 1972, pp. 148-155].

A set structure is significantly different from array and matrix structures since no order is imposed on its constituent data elements. A set is, in a way, the simplest aggregate data structure — all it does is group together a collection of data elements. Sets evolved into programming languages and data base management systems from mathematics where the members or elements of a set usually share a common characteristic and a "characteristic function" is used to define a set. For
PASCAL
MONTHS = (JAN, FEB, MARCH, ...
MSALH : ARRAY [PYEARS, MONTHS] OF REAL;

WHICH ALLOWS EXPRESSIONS SUCH AS
MSALH [1974, JAN]

FIG. 2-26. PASCAL MSALH MATRIX WITH TABULAR INDEXES.

- 69 -
**Matrix**

**Dimension:** 2

**Index Ranges:**
- 1-100 is Employee ID as in ID
- 1000-1999 is Skill Code as in Skillcode

**Semantics**
- **Integer is**
- Years in skill or 0

**Definition**

**Skills Held**

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
<th>1004</th>
<th>...</th>
<th>9999</th>
<th>9999</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fortran**

```fortran
DIMENSION SKHELD (100,1000)
INTEGER SKHELD
```

**Figure 2-27. Fortran ad hoc structure using matrix.**
use in computer science, this function is replaced by the semantics which describe a set structure's purpose. Figure 2-28 demonstrates a set whose elements are aggregates. The rounded sides of the data definition picture indicate that the data elements in a set are unordered. This SKILL set is an alternative to the previous FORTRAN matrix implementation of the same information shown in Figure 2-27. Each instance of the set SKILL contains all the information about one person's experience. This information is composed of n-tuple structures (see below) which record both the SKILCODEs and SKLYRSs. The example Madcap VI statements define the n-tuple ASKIL in the first line and the set SKILLS of up to ten such elements in the second line. Madcap VI does not distinguish between data definition and instance. The statements in Figure 2-28 in effect declare a particular instance of the SKILLS set.

A n-tuple structure is an ordered collection of exactly n data elements. These n elements are referred to either as the first, second, third,...,nth ordinal element or by naming each element. In either case there is an implied order and all instances always have all n elements present. Figure 2-29 describes the 2-tuple (also called an ordered pair) used in Figure 2-28 to record a person's skill by giving the skill and length of experience. Figure 2-29 also shows three alternative ways to define the 2-tuple in Madcap VI. The first line uses named elements, the second explicit numbers, the third implicit numbers; the second and third lines are equivalent in Madcap VI.

N-tuples may be used to structure n pieces of ordered or related data. As another example, suppose AUTHSAL is really three data elements: the minimum, mean, and maximum authorized salary. AUTHSAL could then reasonably be implemented as the 3-tuple shown in Figure 2-30.
DEFINITION

MAXIMUM NUMBER OF ELEMENTS

MAX. SIZE: 10

SET

2 TUPLE OF INTEGERS

DENOTES UNORDERED COLLECTION

DENOTES MULTIPLE OCCURRENCES

1 INSTANCE

MADCAP VI

ASKIL ← (SKILCODE ← I, SKLYRS ← I)

SKILLS* ← \{<0 TO 9> ← ASKIL\}

FIG. 2 - 28. SET.
DEFINITION

2 INSTANCES

MADCAP VI

ASKIL ← (SKILCODE ← I, SKLYRS ← I)

OR

ASKIL ← (0 ← I, 1 ← I)

OR

ASKIL ← (I, I)

FIG. 2-29. N-TUPLE.
MADCAP VI

AUTHSAL $\leftarrow (<0 \ TO \ 2> \leftarrow \text{REAL})$

~ OR ~

AUTHSAL $\leftarrow (\text{MIN} \leftarrow \text{REAL}, \text{MEAN} \leftarrow \text{REAL}, \text{MAX} \leftarrow \text{REAL})$

FIG.2-30. MADCAP VI N-TUPLE FOR A RANGE.
A sequence structure is again an ordered collection, but access is not by ordinal number or index. Instead, only the first element of a sequence can be accessed originally. Then, having done so, the next element and the next element and so on can be accessed one at a time until the end is reached. Thus, with a sequence the index is implicit and access is limited or restricted. A sequence also differs from an n-tuple in that different instances of a certain sequence can have different numbers of elements.

Figure 2-31 shows a sequence for the SALHIST information modeled above as an array structure (Figure 2-22). As indicated in the figure, different instances of SALHIST may record different numbers of previous salaries; the fifth year's previous salary can be found only by a series of operations such as:

First of SALHIST
Next of SALHIST
Next of SALHIST
Next of SALHIST
Next of SALHIST
Next of SALHIST

It should also be noted in Figure 2-31 that VERS2 does not require any upper bound on the number of elements in a sequence.

Some systems provide a slightly generalized version of a sequence which can be accessed in either direction. These systems, instead of just FIRST and NEXT operations, also supply LAST and PREVIOUS operations whose meaning is obvious.
UPPER BOUND ON NUMBER OF ELEMENTS

SEQUENCE

MAX. SIZE: 15 IS PREVIOUS YEARS

REAL IS PREVIOUS SALARIES

SALHIST

VERS 2

SALHIST :: SEQ(REAL)

FIG 2-31. SEQUENCE.
Relations, the final table structure, are more complex than those discussed above. Relation structures also evolved into computer science from mathematics. A mathematical definition is:

Given \( n \) sets \( S_1, \ldots, S_n \) not necessarily distinct, a relation on \( S_1, \ldots, S_n \) is a subset of the Cartesian product \( S_1 \times \cdots \times S_n \).

A Cartesian product is a set of \( n \)-tuples (set and \( n \)-tuple defined similarly to the descriptions given earlier in Section 2.2.2.1) where the \( i \)th element is selected from the \( i \)th set.

The relation-structure which arose from this definition remains quite true to its mathematical heritage. A relation structure and some of the new terms introduced will be illustrated with the example in Figure 2-32. This example details a portion of the PERSON information from the organization data base. This relation is over four sets which are termed "domains". The domain names in this example were picked to imply the characteristics of the four sets. BIRTH, for example, is a date. The domain sets need not be distinct (see following example) so for ease of access, a unique "attribute" name is associated with each use of each domain. An attribute name can occur only once in a relation. Finally, the order of the attributes is unimportant (as indicated by the rounded sides in the data definition picture), except that each \( n \)-tuple making up the relation must be ordered the same way. The one instance of the PERSON relation depicted in Figure 2-32 contains three \( n \)-tuples. The first one indicates that ID 373483014, EMPNAME J. T. Smith, BIRTH October 8, 1947, and AGE 27 are related; i.e., they describe one person.
A UNIQUE NAME FOR EACH SET

PERSON

ATTRIBUTE NAME

ID EMPNAME BIRTH AGE

DOMAIN

ID NUMBER NAME DATE INTEGER

THEN N SETS

ORDER OF ATTRIBUTES UNIMPORTANT

DEFINITION

1 INSTANCE

ALPHA

DOMAIN ID PIC 9(9)
NAME PIC A(VAR)
DATE PIC 99X99X99
AGE PIC 99

RELATION

PERSON (ID, EMPNAME, BIRTH, DATE, AGE) KEY ID

FIG. 2-32. N-ARY RELATION.
Even though attribute and domain names serve distinct purposes, the ALPHA data base management system allows domain names to be used as attribute names if the domain names are unique for a relation. Thus, in Figure 2-32, the attribute name ID is the same as the domain name. ALPHA also allows domain names to be prefixed with another term to ensure unique attribute names. An example of this kind of attribute name is EMP-NAME in Figure 2-32. The concept of "key" in ALPHA is for unique identification of the n-tuples comprizing a relation and is not of importance here.

Another example, represented in Figure 2-33, illustrates the reason for a distinction between attribute and domain names. This relation structure relates a supervisor and his/her subordinates; both domains are identical - all employee ID numbers. However, the two attributes, SUP-ID and SUB-ID must be unique in order to allow reference to either a supervisor or subordinate when processing the data base.

Relations are currently seen as a natural, easy to understand way to record general purpose data (see [Codd 1971]). The relational model for data base management systems is based upon the relation structure shown here restricted to fit certain "normal forms". These semantic restrictions are an important aspect of the relational model and are described fully in the references listed for ALPHA in Appendix A.

Although all relations can be structured as discussed above, it is beneficial to distinguish between n-ary relations with $n \geq 3$ and binary relation structures. Binary relations have a much longer history of use in data base management systems and a very appealing simplicity. The BOSS relation in Figure 2-33 is 2-ary or binary; it is redrawn in a format specialized for binary relations in Figure 2-34. This alternative form characterizes a binary relation as a
ALPHA

DOMAIN ID PIC 9(9)

RELATION BOSS(SUP-ID, SUB-ID) KEY SUB-ID

FIG. 2-33. RELATION WITH TWO IDENTICAL DOMAINS.
FIG. 2-34. BINARY RELATION.
set of "triples". Each triple consists of an "attribute" an "object" and a "value". The attribute corresponds to the relation name in the n-ary relation structure; the object and value represent the two domains. In Figure 2-34, three instances of the BOSS triple as shown; this set of triples stores the same information as in the BOSS relation instance of Figure 2-33.

The LEAP and TRAMP data base management systems ([Rovner and Feldman 1969; Feldman and Rovner 1969; Ash and Sibley 1968]), implemented data bases of triples. Neither of these systems provide any sort of data definition; the triples are created and manipulated with a procedural language. Both systems provide an economical and consistent access and retrieval scheme based upon eight symmetric formats. These eight primitive retrieval forms allow sophisticated questions to be built-up by nesting and the creation of sets to hold intermediate results. This retrieval scheme is quite elegant, and while it is not relevant to a discussion of data structures, it is certainly of interest to information retrieval based upon binary relations (see the three references mentioned above for more information).

2.2.2.2 Group

The two kinds of group structures listed in Appendix A both evolved in data base management systems. Both structures arose from attempts to model data as it actually occurs in the real world. Additionally, the two structures are often confused by both people and systems. As shown in the following discussion, the two concepts involved are really quite different.

A hierarchy structure describes something composed of a group of other data elements. The constituent data
elements are identified by name and are usually thought of as a refinement of the hierarchy element being defined. Figure 2-35 exhibits a BIRTH date hierarchy consisting of three components - a year, month, and day. Thus each instance of a BIRTH structure consists of three components named YEAR, MONTH, and DAY. This simple example illustrates the primary motivation for the hierarchy structure: it allows individual data elements to be grouped together in a natural manner.

Of course it is reasonable to carry on this grouping at more then one level, creating a nested hierarchy structure. Figure 2-36 shows a further refinement of the BIRTH hierarchy; DAY is redefined to consist of a date and the name of the day. Figure 2-36 contains two hierarchy structures - BIRTH and DAYDATE.

All systems which provide a hierarchy structure allow nesting of hierarchies (sometimes placing a limit the maximum number of levels, see Appendix A). This leveling mechanism is provided in two different ways by the system's data definitions. The first way is by naming each (one level) hierarchy and using this name as a component of another hierarchy. This method is illustrated with the Algol 68 example in Figure 2-36; the named hierarchy DAYDATE is used as a component of BIRTH. (Since Algol 68 does not provide tabular basic items, DAYNAME must be declared as an integer.) Alternatively, the entire nested hierarchy structure can be defined together as shown in Figure 2-37. The Algol 68 example in Figure 2-37 nests type definitions (as discussed above in Section 2.2.1.3 and Figure 2-9). The CODASYL DBTG example spells out the entire structure using "level numbers" which appear as a prefix to each basic item. (In CODASYL DBTG integer tabular items are allowed, as with
ALGOL 68

```algon68
STRUCT( INT YEAR, INT MONTH, INT DAY) BIRTH;
```

**FIG. 2-35. HIERARCHY.**
FIG. 2-36. NESTED HIERARCHY.
ALGOL 68

\[
\text{STRUCT} ( \text{INT} \ Y \ E \ A \ R, \ \text{INT} \ M \ O \ N \ T \ H, \ \text{STRUCT} \\
\quad ( \text{INT} \ \text{DATE}, \ \text{INT} \ \text{DAYNAME} \ ) \ \text{DAY}) \ \text{BIRTH} ;
\]

CODASYL DBTG

1 BIRTH.
2 YEAR TYPE IS FIXED 4.
2 MONTH TYPE IS FIXED 2.
2 DAYDATE.
\begin{align*}
3 \text{DATE TYPE IS FIXED 2.} \\
3 \text{DAYNAME TYPE IS FIXED 1; CHECK IS RANGE OF 1 THRU 7.}
\end{align*}

FIG. 2-37. ADDITIONAL DECLARATIONS OF NESTED HIERARCHY.

-86-
DAYNAME in Figure 2-37). All systems using hierarchy structures allow levels using one or both of these methods.

A special version of a hierarchy structure is rarely provided but extremely useful. A hierarchy with alternatives is a refinement with two or more alternative forms (i.e., different constituents). The JOB information in the ORGANIZATION data base pleads for the sort of structure portrayed in Figure 2-38. The information defining a JOB can have two forms: a job currently being carried on by the ORGANIZATION has the three components JOBCODE, AUTHQUAN, and AUTHSAL; a job for which approval is currently being sought has only the two components JOBCODE and APPSTAT. Different instances of the job hierarchy described in Figure 2-38 can have different components (as shown on the right side of the figure). Some components are present in all instances (JOBCODE in this example). Additionally, a special "tag" or identifier is present in all instances; the tag is used to distinguish between the various alternatives. The data definition in Figure 2-38 shows the common components and the tag in its upper block; each alternative is then described with an additional block prefixed by a tag value. Each alternative may have an arbitrary number of completely different components. The tag must be set to properly identify each new instance and can be checked by the user to control program flow according to which alternative is assumed by a particular instance. The hierarchy with alternatives is an ignored data structure of great potential; it provides an excellent model of many real world situations.

Different systems allow access to the component data elements of a hierarchy in different ways. Some of the alternatives for referencing the DAYNAME constituent of the hierarchy shown in Figure 2-36 are:
FIG. 2-38. HIERARCHY WITH ALTERNATIVES.
where the first line allows direct access to the desired element (assuming it is unique) and the others require "navigating" down to the desired element from the top of the hierarchy (using numerous syntaxes). Some systems also allow manipulation of the entire hierarchy by name. For instance, it would be useful to be able to pass the BIRTH hierarchy to a subroutine as a single data element (instead of passing its four components as separate parameters).

A repeating structure is often confused with a hierarchy. A repeating structure collects together a number of data elements of the same type. These data elements can be of any type. For example, SALHIST as defined in Figure 2-22, could have been described as a repeating group of real numbers. However, it is much more common for the elements of a repeating structure to be hierarchies; this is where the confusion begins.

Figure 2-39 demonstrates a repeating structure whose components are hierarchies. The repeating structure SKILLS records all those skills possessed by a person; each single SKILL is described as a two component hierarchy. Data base management systems often merge this distinction between a repeating structure and its components. Many systems, including TDMS as shown in Figure 2-39, assume the repeating structures will always be composed of a hierarchy. In the TDMS declarations shown, SKILLS is a "repeating group" and there is no name for each element of SKILLS. Instead the components of the assumed hierarchy, SKILCODE
LIMIT ON NUMBER OF ELEMENTS, IF ANY.

REPEATING

SKILLS
MAX ELEMENTS: VARIABLE

HIERARCHY IS SKILL INFORMATION

TYPE OF ELEMENTS
UNORDERED

SKILLS

1998 2
1557 3
1002 1
1000 5

1000
1275 15
1276 10

DEFINITION
2 INSTANCES

TDMS

1 SKILLS( REPEATING GROUP)

2 SKILCODE(NUMBER IN SKILLS) VALUES ARE 1000 ... 1999

3 SKLYRS(NUMBER IN SKILLS)

FIG. 2-39. REPEATING STRUCTURE.
and SKLYRS, are listed. (The prefix numbers do not denote levels in TDMS; they are just unique sequential numbers to identify each element of a data base.) To reiterate, a repeating data structure allows multiple data elements to be associated together; in Figure 2-39 the elements just happen to be a hierarchy.

The confusion among systems and humans between hierarchy and repeating data structures is compounded by the fact that repeating structures also commonly occur as members of a hierarchy. Considering the organization data base, PERSON can be thought of as a repeating structure; each element is a hierarchy describing one person. One component of this hierarchy is a repeating structure SKILL; each element is defined as a hierarchy containing information about one skill. The distinction to be illustrated here can be made more clear by redrawing a portion of the organization data base from Figure 1-1. In Figure 2-40, additional names have been added so that there is a single name for each hierarchy and repeating structure. Now, a repeating structure PEOPLE consists of a PERSON hierarchy which contains a SKILLS repeating structure consisting of the ASKILL hierarchy. Appropriate data definitions for this revised data base are developed in Figure 2-41. This final view should make the distinction clear. Unfortunately, the earlier form (Figure 1-1) is much more common in data base management literature and is the basis for the data declarations in many systems.

Returning to consideration of just the repeating structure, one special feature deserves mention. A count field is often provided so that a user can determine the number of elements currently in a particular instance of a repeating structure. This field is a data element which is updated automatically by the system whenever the instance
FIG. 2-40. ORGANIZATION DATA BASE REDRAWN.
FIG. 2-41. DATA DEFINITION FOR REDRAWN ORGANIZATION DATA BASE.

-93-
changes. A declaration of the SKILLS structure from Figure 2-39 is shown in Figure 2-42 with a count field. (The prefix numbers in CODASYL DBTG denote levels in a nested hierarchy structure.) The data element SKILLCNT can be accessed by the user to determine the number of elements in a particular instance of the SKILLS repeating structure. Appendix A notes those systems which supply a count field for repeating structures.

2.2.3 Associations

Associations, or relations, are used to string together data, to order it, and to build complex ad hoc data structures to meet the needs of special applications. The term "relation," although commonly used for such data structures, is avoided here due to the possible confusion between it and the "relational approach" to data management (based upon the aggregate data structure termed the "mathematical relation" in Appendix A). The structures detailed as associations in Appendix A use a variety of methods; due to their heritage, these methods are subdivided into two classes.

2.2.3.1 Pointer Based Association

Pointer based associations arose from ad hoc programming with the pointer items discussed in Section 2.2.1.2. They include many of the classical data structures which programmers have implemented usually within assembly language systems (the methods used are discussed throughout [Knuth 1969, chp. 2]). These data structures are rarely provided explicitly in current programming languages; some are supplied by data base management systems as shown in Appendix A.

A tree structure is defined in the following excerpt from [Knuth 1969, p. 305]:
CODASYL DBTG

1  SKILLCNT;  TYPE FIXED 2.
1  SKILLS;  OCCURS SKILLCNT TIMES.
  2  SKILCODE;  TYPE FIXED 4;
      CHECK IS RANGE OF 1000 THRU 1999.
  2  SKLYRS;  TYPE FIXED 2.

FIG. 2-42. REPEATING STRUCTURE
      WITH COUNT FIELD.
"Let us define a tree formally as a finite set $T$ of one or more nodes such that: a) There is one specially designated node called the root of the tree, root ($T$); and b) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$ and each of these sets in turn is a tree. The trees $T_1, \ldots, T_m$ are called the subtrees of the root."

The term "node" corresponds to the term data element as used here.

Tree structures can be used to represent the interrelation between organizations in the example database; such a tree structure is shown in Figure 2-43. The ORGANIZATION TREE begins with a root which is given the distinguished name DIRECTORS. Trees define levels among their component data elements. Organization 5331 is the first level; 5325 and 5371 form the second level; 5301, 5302, 5303 form the third and final level in the instance shown on the right side of Figure 2-43. Unlike the nested hierarchy structures described in Section 2.2.2.2, each level of a tree has the same format. Additionally, a tree may have an arbitrary number of levels, unlike nested hierarchies in which each level is explicitly declared. This latter distinction is pointed out in [Bobrow 1972] where trees are called "recursive groups" to indicate that they have "arbitrary depth."

The data definition presented in Figure 2-43 tries to capture the generality of several possible kinds of trees. It allows the number and kinds of subtrees and the names used to access them to vary between different nodes or data elements. The example tree shown has a variable number of subtrees for each data element but just one name for all the subtree links. This organization implements the one-to-n nature of SUBORG as shown in Figure 1-1. SUBORG and REPORTO are, in some sense, inverses;
**Fig. 2-43. TREE**

VDL

\[ IS\text{-}\text{ORGANIZATION}(x) = \]
\[ IS\text{-}\text{ORGCODE} \cdot \text{ORGCODE}(x) \]
\[ \& IS\text{-}\text{ORGNAME} \cdot \text{ORGNAME}(x) \]
\[ \& (\exists s_1, \ldots, s_n) (n \neq 0 \& (\forall i)(IS\text{-}\text{ORGANIZATION} \cdot s(i, x))) \]
however, it is convenient to name both of them for use in "navigating" through the tree.

True tree structures are rarely provided explicitly in current systems, as shown in Appendix A. The example tree declaration in Figure 2-43 is drawn from VDL which is not really a programming language. VDL is a language for defining programming language semantics; nevertheless, it has some very interesting data structuring ideas. The VDL "predicate" presented is a close approximation to the tree described by the data definition in Figure 2-43. Assuming proper declarations for the subpredicates (IS-ORGCODE and IS-ORGNAME) the predicate IS-ORGANIZATION is satisfied by VDL "objects" such as the one shown in Figure 2-44.

As mentioned above some tree definitions require that every data element (except those at the final level) in every instance have the same number of subtrees. In this case, "binary" trees which have two subtrees for each data element are of interest. Binary trees are sufficient to store the information of any more general tree - if some form or organization can be sacrificed (see [Knuth 1969, pp. 332-345]).

Tree structures form a very natural hierarchy of information. They are widely used, for instance in language parsing as "syntax trees" [Gries 1971, chp. 2]. Trees have been widely implemented using other data structures including arrays and owner-member relations; thus, they are widely used despite the few systems offering them explicitly.

Linked list structures are also often used in ad hoc programming in assembly language. ([Knuth 1969, chp. 2] is again the primary source for details of such methods.)
FIG. 2-44. VDL TREE OBJECT.
linked list structure relates a collection of data elements by specifying the actual connections used to order them in a sequential manner. Again borrowing a definition from [Knuth 1969, p. 234]:

"A linear list is a set of \( m \geq 0 \) nodes ... whose structural properties essentially involve only linear (one-dimensional) relative positions of the nodes..."

There are again many variations. Linked lists may be connected in one or two directions (often termed "forward" and "backward"). There may be a designated starting point called a "head cell". If a head cell is used, every element may be linked directly to it. The links may be "circular"; i.e., the last data element connected back to the first or head cell. The data definition form presented in Figure 2-45 includes provision for most popular variations. However, there are so many ways to implement linked lists that some modification to this data definition may be necessary to model the structures provided by some systems. Additionally, a truly generalized linked list structure, as suggested by all the alternatives shown in this data definition, does not exist in any single language.

The SKILLS linked list of Figure 2-45 groups together the skills held by a person. To find all the skills a person has or to see if a person has a particular skill the NEXTSKILL links are followed from the head cell through each element.

The CODASYL DBTG declarations in Figure 2-45 create what is called a "set" in DBTG terminology. This "set" is really more complicated than a linked list structure; the feature of interest here is the "mode is chain" clause which creates a linked list with a head cell SKILLHEAD. This
ONE LINK IS MANDATORY

LINKS:
FORWARD: NEXTSKILL
BACKWARD: NO
HEADCELL: NO
HEAD CELL: YES, SKILLHEAD
TAIL CELL: NO
CIRCULAR: YES

HIERARCHY IS
SKILL INFORMATION

LINKS MAY BE NAMED

CODASYL DBTG

RECORD NAME IS SKILLHEAD ...
RECORD NAME IS SKILL ...
1 SKILCODE; TYPE FIXED 4.
1 SKLYRS; TYPE FIXED 2.

SET SKILLS;

MODE IS CHAIN;
ORDER IS LAST;
OWNER IS SKILLHEAD.
MEMBER IS SKILL.

FIG. 2-45. LINKED LIST.
A directed graph structure is similar to a linked list; it also relates a collection of data elements by specifying connections or links but the elements need not be sequentially ordered. In [Knuth 1969, p. 312] directed graphs are called "Lists" and defined as follows:

"A List is a finite sequence of zero or more atoms or Lists."

The connections between elements in a directed graph are completely arbitrary; connections may overlap and form loops through the data.

A possible general data definition form is shown in Figure 2-46 where a data structure to represent which ORGANIZATIONs pass reports to which other ORGANIZATIONs is portrayed. The links pictured in the instance of ORGANIZATION NET show which directions reports travel; for example, 5303 sends reports to 5331 and 5371 while receiving them from 5325 and 5302. This possible bureaucratic nightmare is an example of the complex data structures which can be created using directed graphs.

Complete directed graph structures are seldom provided in systems; however, Madcap VI is an exception. The instance of ORGANIZATION NET shown in Figure 2-46 is created from Madcap VI statements at the bottom of the figure. (Madcap VI does not provide a data definition capability.) The
SOME LINKS REQUIRED FOR EVERY ELEMENT

DIRECTED GRAPH

REQUIRED LINKS: NONE
OPTIONAL LINKS:
REPORTO IS WHERE REPORTS ARE SENT
SUBORG IS WHERE REPORTS ARRIVE FROM

HIERARCHY IS ORGANIZATION

MAY HAVE ONE OR OTHER LINKS
MORE KINOS OF ARE OPTIONAL
ELEMENTS

ORGANIZATION NET

DEFINITION 1 INSTANCE

MADCAP VI

A ← (ORGCODE ← 5331, ORGNAME ← "SOFTWARE R & D")
B ← (ORGCODE ← 5325, ORGNAME ← "SOFTWARE PRODUCTION," REPORTO ← {A,F})
C ← (ORGCODE ← 5371, ORGNAME ← "SOFTWARE TESTING," REPORTO ← {A})
D ← (ORGCODE ← 5301, ORGNAME ← "XYZ PROJECT," REPORTO ← {B})
E ← (ORGCODE ← 5302, ORGNAME ← "ABC PROJECT," REPORTO ← {D,F})
F ← (ORGCODE ← 5303, ORGNAME ← "SUPPORT," REPORTO ← {A,C})

FIG. 2-46. DIRECTED GRAPH.
A wide variety of directed graphs have been successfully used. One example is the "ring" [Gray 1967], also called a "hierarchical structure" [Williams 1971], which is similar to a hierarchy of linked lists. The field of graph theory has developed a mathematical formalism for "directed graphs," distinguishing "cyclic" and "acyclic" graphs (for instance, [Pfaltz 1972]).

2.2.3.2 Other Associations

Some other associations have evolved which relate data elements without need for explicit pointer or pointer-like items. These relations are grouped under the heading of other associations in Appendix A.

The owner-member structure relates a collection of data elements in which there is one distinguished element termed the "owner". All the other elements are subordinate to the owner and are called "members". Figure 2-47 describes an owner-member association named PEOPLE which records the various PERSONs in an ORGANIZATION. The association defined is between the ORGANIZATION hierarchy, which is the owner, and any number of member PERSON hierarchies. Although PEOPLE is composed of data elements of two types (both hierarchies), this is not a necessary restriction. An owner-member association can have more than one type of member; alternatively, the owner and members can be of the same type.

The CODASYL DBTG "set" mentioned above as an example linked list structure is also an owner-member association.
COULD LIMIT SIZE OR REQUIRE SAME NUMBER IN EACH INSTANCE

OWNER-MEMBER ASSOCIATION

NO. MEMBERS: VARIABLE

PEOPLE IS PERSONS IN AN ORGANIZATION

ONE OWNER MANY MEMBERS, POSSIBLY OF DIFFERENT KINDS

DEFINITION

CODASYL DBTG

RECORD NAME IS ORGANIZATION ...
  1 ORGCODE ; TYPE IS FIXED 4 . ...

RECORD NAME IS PERSON ...  
  1 ID; PICTURE "9(9)".

SET NAME IS PEOPLE ;  
ORDER IS LAST ;  
OWNER IS ORGANIZATION ;  
MEMBER IS PERSON .

FIG. 2 - 47. OWNER-MEMBER STRUCTURE .
Both owner and member data elements must be "records" (a combination of repeating and hierarchy structures). Figure 2-47 also shows the PEOPLE structure declared according to CODASYL DBTG. The two kinds of component elements, ORGANIZATION and PERSON "records", must be defined, then associated with the "set" declaration PEOPLE. This example clearly illustrates that the CODASYL DBTG "set" is completely alien to the more basic set structure discussed in Section 2.2.2.1.

Functional association evolved from the mathematical concept of function. A function "maps" or relates the elements of a set called the "domain" to the elements of another set termed the "range". Each domain element is associated with exactly one range element. Thus, a functional structure is an association between two collections of data elements which relates each member of the first collection to some element of the second collection. The mathematical concepts of "one-to-one," "onto," and "inverse" functions (as defined in any introductory calculus text) may also be useful in describing a functional association and should be included in the data definition when applicable.

Figure 2-48 shows a functional association relating people to their skills. The range of this association is sets of n-tuples describing a skill and number of years experience. Thus, the SKILLSHELD association maps each PERSON to a set named SKILL (as defined in Figure 2-28). The range of SKILLSHELD is the collection of all sets which describe any person's experience. Figure 2-48 shows the declaration of the SKILLSHELD function in VERS2 along with declarations of its domain and range.
FIG. 2-48. FUNCTIONAL ASSOCIATION.
The functional structure is appealing because it is very abstract; it does not imply any particular implementation. In fact, this function could be implemented as a matrix structure (as in Figure 2-27) or as a nested hierarchy of repeating structures (as in Figure 2-41). Similar methods of abstraction to postpone implementation decisions are the basis of currently popular "top-down design" methods (see Section 4) and the "implementation facility" proposed for programming languages in [Earley 1971].

2.2.4 Files

Files are major components of a data base. They group together basic items, aggregates, and their associations adding still more structure to determine how the file will be organized. Some file organizations are provided by programming languages and data base management systems; others are provided as integral parts of computer operating systems. This later kind of file is not covered here since they tend to be specialized to a particular computer system.

2.2.4.1 Arbitrary Access Algorithm

Arbitrary access algorithms are in a class by themselves in Appendix A since, although they can implement file structures, they can also be used to define any of the data structures described in all of Section 2.2. Arbitrary access structure gives total freedom to the user to define any type of access whatsoever, usually in terms of some programming language. Since infinite variations are possible, a data definition is an impossible task.

Arbitrary access algorithms are provided by two of the systems tabulated in Appendix A. The programming language
BLISS allows "structure" definitions which can include anything possible in the language. These operations are, in effect, grouped together and given a name by the "structure" which then acts like a shorthand or macro. (A full description of this concept is in [Wile and Geschke 1972].) In BLISS, all data structures are provided in this way; the only built-in data organization is an array which is implemented as the "default structure".

The MacAIMS "relational strategy module" is a user-written program which allows completely arbitrary file structures. The file organizations provided in this way are converted to a standard "canonical form" to interface with the rest of the system. The "relational strategy module" does this as well as storing and manipulating the file.

The advantage of both these schemes is that all implementation details are consolidated in a single place (i.e., the definition of the arbitrary access algorithm) instead of propagated throughout all the user programs and procedures. The access algorithms are referred to by name whenever needed; but, their details do not clutter up the scene. The disadvantage of such a scheme should be obvious. While the user has complete freedom to do whatever the language allows, he/she is burdened by the necessity to invent new access algorithms whenever a new need arises.

2.2.4.2 Indexing

All the other file structures provide methods for indexing or selecting from the collection of data elements comprising the file. The reader may be disappointed to note that two traditional file structures, the inverted and multilist files, do not appear in Appendix A. This
omission is because no programming language or data base management system provides these structures for use by their users. Both these files are used as an implementation basis for data base management systems. TDMS, for instance, creates a fully inverted file (called a "concordance", see [Bleier and Vorhaus 1969]) for all basic items in the user's data definition. The resulting structure is not under control of the user; instead, the system uses it to process retrieval requests.

At any rate, numerous descriptions of inverted and multilist files have appeared ([Dodd 1969] and [Lefkovitz 1969] are good treatments). Additionally, an excellent formalism for these organizations has been developed ([Hsiao and Harary 1970], discussed in Section 5.1.3).

The three indexing structures listed in Appendix A are provided for the user's benefit by programming languages and data base management systems. They all impose a file structure on a collection of data elements.

A key file is accessed with a special tag or identifier called the "key". The data structure associates one or more data elements with each key. The term key is thought of in the sense of "the key to a problem". By specifying a key value, all the associated data elements in the file can be accessed. The two main kinds of key files are distinguished by the sort of quantity which can make up a key.

If a key can be any basic item, then any value of the item is a suitable key. This kind of key file is shown in Figure 2-49 where a PEOPLE file to record information about an organization's people is postulated. The key to this file is the basic item ID within the PERSON hierarchy structure as shown in Figure 2-41. Each value of the ID item
DEFINITION RETRIEVAL WITH KEY OF '373483014'

1 INSTANCE AND EXAMPLE RETRIEVAL WITH KEY OF '373483014'

CODASYL DBTG

RECORD NAME IS PERSON;
LOCATION MODE IS CALC USING ID
 DUPLICATES ARE NOT ALLOWED;
 WITHIN PEOPLE-AREA.

1 ID; PICTURE IS "9(9)."
1 EMPNAME; ...

FIG. 2-49. KEY FILE.
is a suitable key and can be used to retrieve the hierarchy structure instance with matching value from the file.

Key files with arbitrary basic items as keys may be implemented in a variety of ways. They may use "hashing" schemes to map the possible values of the key item into a small range of integers suitable to identify elements of the file. Alternatively, they may be based upon multilist or inverted file structures as discussed at the start of this section. Hashing methods are described in general in [Knuth 1973, chp. 6.4] and in reference to information retrieval in [Price 1971].

Figure 2-49 shows a key file declaration in CODASYL DBTG. This declaration will be based on hashing as specified by the "location mode is calc" clause. A system-supplied "standard randomizing routine" will be called automatically every time the PEOPLE file is to be accessed. The "duplicates are not allowed" clause rules out two occurrences of data element PERSON with the same ID in any single key file.

An alternative kind of key file is shown in Figure 2-50, again for a PEOPLE file. In a special key file the system works from a special quantity which is created by the system and saved by the user for retrieval use. Only these special values are allowed as keys to such a file. This special quantity may be called a "record number", "reference number", or "data base key". In all cases, it is built by the system and returned to the user when a data element is first stored. This key can be used at any later time to retrieve the original data element. The contents and format of these special keys need not be known to the user.
FIG. 2-50. SPECIAL KEY FILE.

- 113 -

RECORD NAME IS PERSON,

LOCATION MODE IS DIRECT PERSON-KEY,
WITHIN PEOPLE-AREA.

1 ID, PICTURE IS "9(9)."
1 EMPNAME, ...

CODASYL DBTG
Figure 5-50 details a special key file using COPASYL DBTG. The "direct" file can be accessed by initializing PERSON-KEY to the "data base key" of the desired record. This key uniquely identifies an element of the file so the "duplicates" clause used in Figure 2-49 is not needed here.

A current pointer file structures a collection of data elements so that, at any given moment, one or more of the data elements are thought of as being the current centers of attention within the file. Manipulation of the file can be based upon one of the current pointers. The various pointers are usually independent from one another, but, in some cases, it is useful to have one pointer "follow" another through the file. Given a current pointer indicating a particular element of a file, the user needs some way to set or move this pointer. There are two possible methods: explicit commands which change a current pointer, or implicit updating of the pointer whenever some non-current pointer operation is performed.

A possible pointer file data definition for the same PEOPLE file considered above is shown in Figure 2-51. There is only a single pointer named CURRENT; it selects the particular instance of the PERSON hierarchy most recently accessed. IDS maintains a single current pointer for each type of "record" (a repeating structure). Some commands to use this pointer are shown at the bottom of Figure 2-51. These commands assume the current pointer is already set up. (In IDS, the pointer is updated automatically by the system whenever certain other operations are performed; this implicit operation is not under user control.) The first two commands shown do not modify the current pointer; it still selects the same data element afterwards. The
PEOPLE

CURRENT POINTER FILE

NO. OF POINTERS: 1
CURRENT IS LAST ELEMENT
ACCESSED

HIERARCHY
PERSON

PEOPLE

CURRENT

T.B. HENRY

223253124
L.A. JONES

373483014
J.T. SMITH

365567378
A.S. WILSON

EXAMPLE FILE ACCESS COMMANDS:

MODIFY CURRENT PERSON RECORD, REPLACE CURSAL FIELD
GET CURRENT PERSON RECORD
DELETE CURRENT PERSON RECORD

FIG. 2-51. CURRENT POINTER FILE.
third command wipes out the current pointer; afterwards there is no current pointer until it is reestablished using some other sort of access.

The CODASYL DBTG provides a whole collection of current pointers, the status of which is changed automatically by other commands. However, the user can save the current pointers and restore them at some later time. Thus, CODASYL DBTG includes both explicit and implicit pointer modification. These pointers are not declared by the user's data definition; instead, they are always provided and maintained by the system.

The final data structure listed in Appendix A is also the oldest file organization. A sequential file structures a collection of data elements into a strictly ordered sequential stream according to the value of some component of each element. A sequential file is always accessed in the same order; particular data elements are located by finding their place in the ordering. Such files provide no "direct" or "random" access to a particular data element (in contrast to both key and current pointer files).

Figure 2-52 presents the PEOPLE file once again as a sequential file ordered by the ID component of the PERSON hierarchy. This kind of file structure evolved from consideration of computer card decks and tape reels for which only sequential access was possible. However, there is no need to restrict the sequential file to any particular implementation; it should be used wherever it is the correct data structure to model the data at hand. Figure 2-52 illustrates an implementation-independent sequential file declaration in PASCAL.
PEOPLE: FILE OF PERSON;

EXAMPLE FILE ACCESS COMMANDS:

GET (PEOPLE)
RESET (PEOPLE)

FIG. 2-52. SEQUENTIAL FILE.
2.3 Observations

The preceding section has surveyed the data structures furnished by popular data base management systems and programming languages. These data structures and the systems which supply them are tabulated in Appendix A.

This collection of data structures forms a complete set, in a pragmatic sense, since they are the data organizations provided by 21 popular systems. Thus this collection of data structures is a valuable guide for the design and comparison of both languages and data base management systems. It will be used, in Section 6.1, to demonstrate that the data structure model introduced in the following section can represent the collection of common data structures. It is also valuable to programmers using any language because it provides a model for them to use when selecting data organizations. The programmer can pick, from among this collection of data structures, the organization best suited to the task at hand. Then the selected structure can be implemented using whatever facilities the particular language furnishes. In this way, the programmer is relieved from constantly creating new ad hoc data organizations.

One of the first conclusions possible from Appendix A is that there is considerable overlap between the data structures in programming languages and data base management systems. Perhaps this represents a drawing together of the two fields, at least in the area of data structures. The work of the CODASYL Data Base Task Group certainly is relevant to both fields. In any case the more sophisticated data structures, such as set and hierarchy structures, no longer belong solely to data base management; they are now offered directly to the programmer by the language compiler. Likewise, almost every traditional language data
structure (e.g., matrix) is provided by some data base management system. Any need for distinguishing programming languages and data base management systems when considering data structures is clearly past. Thus it is reasonable to study this wide range of data structures with a single, unified model.

Despite this overlap between programming languages and data base management, there are some useful data structures in Appendix A which have been ignored by most system designers. Both tabular and virtual basic items are so clearly useful in modeling real word data (see Section 2.2.1.3), it is indeed surprising they exist in so few systems.

Another interesting observation from Appendix A concerns the controversy between the network model and relational model for data base management. (This controversy was briefly mentioned in Section 2.1.) If the columns of Appendix A for the CODASYL DBTG (the typical example of a network system) and ALPHA (designed by one of the principle proponents of the relational model) are compared, it is clear that ALPHA provides considerably fewer data structures than does CODASYL DBTG. Thus a user must be aware of more kinds of data structures and be concerned with picking between numerous alternatives when using CODASYL DBTG. On the other hand, ALPHA provides only a few structures, none of which are alternatives of others. This difference is the basis for the "simplicity" argument in favor of the relational model (as expressed in [Date and Codd 1974]).

It is also evident that many different aspects of data structures are combined and offered as a single feature in some systems. An example of this is the widespread confusion between the repeating and hierarchy structures,
as discussed in Section 2.2.2.2. Another example is the CODASYL DBTG set which includes many different data structures.

This combination of data structures may be good or bad. If the combinations which are provided by a system are well suited to the data base being implemented, it will be easier for the user to pick a felicitous data design. However, it is certainly clear that this combination confuses the literature of the field. Although there are only approximately two dozen data structures identified by Appendix A, it contains many more names for the data structures and the combinations. For example, who would think that "records", "structures", "sequences", "classes", "group items" and "repeating groups" are all names for hierarchy structures (or some combination including the aspects of a hierarchy)? Perhaps the terminology presented in Appendix A can be the basis for consistent naming of data structures.

However, as noted in Appendix A, the purpose of that large table and classification exercise is not simply to introduce another set of names for the common data structures. Instead, Appendix A serves as the primary input to the development of a new model for data structures. This model, which does not depend on descriptive names, is introduced in Section 3.
3. A MODEL FOR COMMON DATA STRUCTURES

The previous section has surveyed the mélange of data structures provided by a wide collection of programming languages and data base management systems and has also pointed out some of the confusion which exists because of the way data structures are named. This section develops a model for data structures which is sufficient, both to cover the wide range of data structures and to remedy this confusion.

Section 3.1 introduces the style and form of the data structure model and recounts the motivation which suggested this particular style. Sections 3.2, 3.3, and 3.4 define and illustrate the portions of the model relevant to aggregate, association, and file data structures. Section 3.5 summarizes what has been attained.

The data structure model is the heart of this thesis. It is the basis of the top-down design methodology developed in Section 4. It is compared to other models in Section 5 and arguments for its usefulness are presented in Section 6.

3.1 Motivation

Consider arrays, sets, sequences, n-tuples and matrices. They all share a common purpose or intent: they group together a collection of more basic components. These constituents are in effect gathered together and associated with each other by a data structuring technique. These data structuring techniques vary from programming language to programming language. However, it seems there are some limits to the variations possible within the class of data structures mentioned above. Thus, it seems reasonable to consider a class of data structures which in some general sense serve a common purpose and attempt to extract from
them an understanding of what makes one particular data structure different from others.

Consider arrays as currently provided by popular programming languages. Figures 3-1 through 3-3 show three uses of data structures which might be called arrays; each figure shows both the declarations of the array structure in a current programming language and a pictorial representation of two instances of the data structure. Figure 3-1 shows a traditional array as it might be expressed in FORTRAN [American Standards Association 1966]. The example is drawn from the organization data base shown in Figure 1-1; SALHIST records an employee's five previous salaries. No one would quarrel with the claim that Figure 3-1 represents an array.

Some people would, however, complain when the data structure of Figure 3-2 is called an array. The SKILLS data structure records an employee's experience as a collection of skills and years of experience in each skill. Thus each element or constituent of SKILLS is itself composed of two subelements (named SKILCODE and SKLYRS in Figure 3-2). The VERS2 declarations [Earley 1973] shown define SKILLS as a "sequence" of SKILL data elements; the VERS2 sequence allows, as one alternative, references to its constituents via ordinal number and is thus VERS2's version of the traditional array. However, VERS2 allows any data type whatsoever to occur as elements of a sequence and it permits instances of a sequence to have unequal numbers of elements. For these reasons, some people would feel the VERS2 sequence is not a true array.

Figure 3-3 depicts a data structure which is also part of the organization data base; the data structure PEOPLE contains assorted information on each employee (including
FORTRAN

```
DIMENSION SALHIST(5)
REAL SALHIST
```

```
1 1333.33
2 1095.00
3 925.00
4 675.33
5 600.00
```

```
1 1025.00
2 912.00
3 875.12
4 0.0
5 0.0
```

FIG. 3-1. FORTRAN ARRAY.
VERS2

SKILL :: < SKILCODE $\rightarrow$ INT, SKLYRS $\rightarrow$ INT >

SKILLS :: SEQ(SKILL)

1 <1000, 5>
2 <1002, 1>
3 <1557, 3>
4 <1907, 1>
5 <1998, 2>

1 <1000, 1>
2 <1001, 1>
3 <1021, 5>

FIG. 3-2. VERS2 ARRAY.

-124-
PASCAL

ID = 66001 .. 66999

SKILL = RECORD
  SKILCODE: 1000 .. 1999;
  SKLYRS: INTEGER
END;

PERSON = RECORD
  EMPNAME: ARRAY [1..20] OF CHAR;
  SKILLS: ARRAY [1..10] OF SKILL;
  SALHIST: ARRAY [1..5] OF 400.0 .. 2500.0
END;

PEOPLE = ARRAY [ID] OF PERSON;

ID = 66001

| J.T.SMITH | 1<1000,5> | 1 1333.33 |
| 2<1002,1> | 2 1095.00 |
| 3<1557,3> | 3 925.00  |
| 4<1907,1> | 4 675.33  |
| 5<1998,2> | 5 600.00  |

ID = 66002

| L.A.JONES | 1<1000,1> | 1 1025.00 |
| 2<1001,1> | 2 912.00  |
| 3<1021,5> | 3 875.12  |
| 4<1907,1> | 4 675.33  |
| 5<1998,2> | 5 600.00  |

FIG. 3-3. PASCAL ARRAY.
SALHIST and SKILLS which have just been considered as independent data structures in Figures 3-1 and 3-2). Most anyone will agree that there is more to this data structure than is normally implied by the term array. Not only does the PEOPLE array consist of other than atomic elements, but some of its elements are also arrays. PASCAL allows this nesting of "arrays" as shown in Figure 3-3. The array PEOPLE consists of 999 instances (one per ID) of the data type PERSON which itself contains arrays.

The wide variation between languages in the case of arrays is also present with regard to many other data structures. This wealth of data structures is hard to characterize using individual names for each distinct technique of data organization. Additionally, so many names have already been used that introducing new names just aggravates the confusion. As it is now, one language's "array" (FORTRAN) is another's "sequence" (VERS2) or "table" (COBOL).

A better approach is to distill the major differences or variations out of the current wealth of data structures. In this way, the possible variations or "degrees of freedom" can be considered independently from any particular data structuring technique. One way to do this is to enumerate the degrees of freedom as a set of questions; each question characterizes one possible axis of variation among data structures. The following sections describe such a characterization for each category of data structures from Appendix A.

Before considering the first section of the model, some additional philosophy must be reviewed. The following discussion is meant to define the aims of the data structure model.
First, the purpose of the data structure model is to portray the structural aspects and characteristics of real world data structures. The rules for changing a data structure as it is accessed over a period of time are not specifically included. The model's purpose is to describe the static, unchanging nature of data structures. As will be seen, this focus allows data structures to be modeled quite well. The distinctions between structure and access oriented views of data organization (as well as other approaches) will be developed further in Section 5.

However, data base management does not break down into a black and white distinction between structure and access. In fact, questions of data integrity and protection seem to fall somewhere in between; they add restrictions to a data base's structure in order to control its access. The form of the model developed here does allow some insight into data integrity. In many cases, slight extensions of the model enhance its ability to describe real world information including data integrity constraints. These extensions to the data structure model (marked with an asterisk throughout) may be considered as optional parts of the model, to be answered or not depending on whether data integrity is being considered. Such an extension of the model seems worthwhile because data integrity is an issue of current concern (see Section 2.1), and because it fits within the framework of the model quite well.

Another concept which is both of current interest and relevant to the data structure model is the distinction between logical and physical data structures. This distinction is usually developed as part of an approach to data independence (see Section 2.1). The model proposed here considers only logical data organizations. It seems to be clear that a data structuring technique exists independently of its
implementation in any particular manner. For example, the term "pointer" may imply a certain implementation quite forcefully, but there are certainly many different ways of implementing the same concept. This concept could perhaps better be expressed with one of the following terms: connection, access path, or link. The relationship between the model and any possible implementation aids will be discussed further in Section 7.1.

The data structure model is concerned with logical, conceptual data structures. Thus an unordered "set" of elements may be described as being truly unordered with the model regardless of the fact that most implementations will have to store the elements sequentially. Since the model's chief purpose is to further the understanding of data structures, it is of paramount importance that we concentrate on the conceptual data structure and ignore extraneous details which are assumed by implementations. As another example, an array can be modeled without assuming it is stored contiguously in memory. The guideline should be to make sure that no additional, unnecessary organization is imposed on the data structure under consideration.

The data structure model takes the form of a separate set of axes or questions for each of the three main categories of nonbasic data structures: aggregates, associations, and files. One set of questions with specific answers describes a particular data structure. However, when an actual data base is described, some additional information is required (e.g., name of the data element being defined). In this case, the model plus the additional information is presented in a pictorial style similar to that which was used for the pictorial data definitions throughout Section 2. Although these two approaches are both pictorial, they must not be confused. The drawings of Section 2 simply presented
data definition information in a form similar to traditional data base management systems. The figures here in Section 3, however, present data definitions in terms of a new, axis-oriented approach.

No model is proposed for the data organization techniques classified as basic items in Appendix A. The hierarchical breakdown shown for basic items in the appendix is sufficient to understand them. (The basic items were explained in terms of this breakdown in Section 2.2.1.) For the majority of the basic items, each one is a separate kind of data organization and a sufficiently general model is of little practical use. The hierarchical classification shown in Appendix A is original; it serves as a model for the basic items.

The following sections discuss in detail the three sets of axes which form the data structure model. Section 3.2.1 includes a full description of the pictorial data definition style used. The organization, scheduling, and decision table data structures which were introduced in Section 1.3 are all used as examples in the following discussions.

3.2 Aggregate Model

Aggregates are the first composite data structures discussed in Section 2. An aggregate groups together a collection of separate data elements into a single table-like structure. Examples of each of the aggregate data structures have been discussed in Section 2; the various sorts of arrays shown in Figures 3-1, 3-2, and 3-3 are all aggregates. This section develops a set of questions which characterize the aggregates.

An aggregate data definition lists the kinds of data elements and describes how they are grouped together. An
aggregate instance is an acceptable number of instances of the various elements correctly grouped together. The purpose of the model is to describe the definition of aggregates; it can be viewed as a specification of the rules for creating all possible instances of an aggregate structure.

3.2.1 Model for Aggregates

Figure 3-4 describes the model for aggregates; it lists the five axes in the form of questions and gives a short description, an abbreviation, and sample answers for each axis. Each axis will be discussed below; however, an introductory example will show how they can be used. The best known example of an aggregate is the array - it provides structure via indexing for a fixed sized collection of numbers. Such an array structure could be modeled as follows (using the abbreviations from Figure 3-4):

<table>
<thead>
<tr>
<th>Axis</th>
<th>Description</th>
<th>Abbreviation</th>
<th>Sample Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Homogeneous:</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Basic items:</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ordered:</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>FIXED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NUMBER</td>
<td></td>
</tr>
</tbody>
</table>

This example defines a bare array; to actually define a particular array structure two additional sorts of information must be specified. This additional information, together with the five axis values, is presented as a pictorial data definition as shown in Figure 3-5. The first kind of additional information is the kind or type of the aggregate's elements. This information is shown in the bottom of the data definition block. In the SALHIST example of Figure 3-5, (which represents the same traditional array discussed in Section 3.1 and shown in Figure 3-1) the elements are all of the one type ("Homogeneous" YES) shown -
1. Are the elements homogeneous? Are all instances of the elements drawn from the same data definition? Homogeneous: Yes, No

2. Are the elements basic items? Are all instances of the elements atomic and indivisible? Basic items: Yes, No

3. Are the elements ordered? Is any ordering among the element instances imposed or implied by the structure? Ordered: Yes, No

4. What is the number of elements? How many instances of each kind of element are combined in one instance of the aggregate? Number: Fixed, limited, unbounded

* When "homogeneous" is no, "number" may be extended to specify a different count for each kind of element.

5. How is an element identified? How is an individual element instance named, labeled, or identified within an aggregate instance? Identification: Number, name, pointer, none

* Data integrity

Fig. 3-4. Model for aggregates.
FIG. 3-5. MODEL FOR ARRAY OF FIGURE 3-1.
reals representing previous salaries. The element's type
description is enclosed within an inner box to reinforce the
idea that multiple instances of that type are present in each
array instance.

The other kind of additional information provides
further, specific details to the answers to the five questions.
For example, in Figure 3-5 the answer to the "Number" question
specifies that a fixed number of element instances occur
in each array structure; the additional information indicates
that this fixed number is five. The examples shown
throughout this section show other ways this additional
information is used.

The distinction between what aspects of a data structure
are described by the axes and the additional information can
be further understood by drawing an analogy between tradi­
tional data structure declarations and the model. In normal
programming languages, a data structure called an array is
provided; to use an array the programmer specifies, in a
declaration statement, the dimension of the particular array
being defined. No matter whether the dimension is 5 or 2000,
in either case the same basic kind of data structure is being
defined. Likewise, the questions of the model proposed here
define a basic kind of data structuring technique. The addi­
tional information provides the details, parameters, or
specifics for a particular use of the data structuring tech­
nique. Thus, as shown in Figure 3-5, the axes model the
concept of "array" and the additional information provides
the counterpart to the traditional declaration's dimension.

A few other conventions of the pictorial data definition
deserve mention. The data structure definition is given a
name so that it may be referred to conveniently and used in
other definitions; this name appears on the left side of
the data definition block. A traditional or descriptive name for the data structuring technique modeled may optionally appear above the block. This name is added for convenience only; due to the confusion between these names, as discussed in Section 3.1, the model should be the basis for any serious consideration of the data structure. Finally, the special notation "is" precedes semantics or commentary information about the data structure.

Now each axis will be explained. Examples will be drawn from the decision table discussed in Section 1.3 and from Figures 3-1 through 3-3.

The traditional array as modeled above contains data elements of exactly one kind or type; however, other aggregate structures need not consist of like elements. The hierarchy structure of Appendix A usually contains nonhomogeneous elements. For instance, each row of the decision table in Figure 1-5 can be thought of as a two element hierarchy consisting of a set of conditions and an action. In general, a hierarchy can be modeled:

- Hierarchy: NO
- Basic items: NO
- Ordered: NO
- Number: FIXED
- Identification: NAME

A pictorial data description for DTROW, one row of a decision table, is shown in Figure 3-6. In this case, the "Homogeneous" question is answered NO; the elements which make up one row of the decision table are of two different kinds, called CONDITIONS and ACTION. On the other hand, the SKILLS array of Figure 3-2 is homogeneous; each element is the same: a data element of type SKILL. A pictorial data definition
FIG. 3-6. MODEL FOR ONE ROW OF DECISION TABLE.
for this array is shown in Figure 3-7. The SKILLS array itself is shown at the top of Figure 3-7; a definition for its elements is shown at the bottom of the figure. The elements, of type SKILL, are also aggregates. In this definition "Homogeneous" is answered NO for reasons of data integrity. Although both SKILCODE and SKLYRS are integers, there may be various restrictions on their values (as implied in Figure 3-2). In this case, the "tabular" basic item of Appendix A is an appropriate choice for the definition of SKILCODE and SKLYRS. Such a definition for SKILCODE is shown in the PASCAL example in Figure 3-3. Thus, the "Homogeneous" axis does not distinguish between atomic and complex data elements; it simply notes whether or not all the instances of the aggregate's elements are drawn from the same definition.

The distinction between atomic and complex constituents is made by the second question. The term "Basic items" arises from Appendix A where it used to denote data which is usually not divisible. The array of Figure 3-1 has only basic items (of one certain kind - regular numbers) as its elements. This array is modeled in Figure 3-5. However, many programming languages allow arrays of arbitrary data types, including other aggregates. The entire decision table of Figure 1-5 could be an array of the hierarchies mentioned in the preceding paragraph and shown in Figure 3-6. A data definition in terms of the model for this array is shown in Figure 3-8; its elements are defined by referring back to the definition given in Figure 3-6. The PEOPLE data structure of Figure 3-3 does not consist of basic item elements; each element in the rather complex data structure is given the name PERSON in the figure. A model of this data structure is shown in Figure 3-9. The constituents of the SKILLS array in Figure 3-2 are also not basic items, although the SKILL type is somewhat simpler than PERSON.
### Skills

- **Array**
  - Homogeneous: Yes
  - Basic Items: No
  - Ordered: Yes
  - Number: Unbounded
  - Identification: Number, 1 - N

Skill is person's experience in one skill

### Hierarchy

- Homogeneous: No
- Basic Items: Yes
- Ordered: No
- Number: Fixed, 2 (one of each)
- Identification: Name

<table>
<thead>
<tr>
<th>Skill Code</th>
<th>Sklyrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer is skill code</td>
<td>Integer is years in skill</td>
</tr>
</tbody>
</table>

**Fig. 3-7.** Model for data structure of figure 3-2.
FIG. 3-8. MODEL FOR ENTIRE DECISION TABLE.
PEOPLE

ARRAY

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: YES
NUMBER: FIXED, 999
IDENTIFICATION: NUMBER, 66001-66999

PERSON IS ALL THE INFORMATION ON A SINGLE PERSON

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 3
IDENTIFICATION: NAME

PERSON

<table>
<thead>
<tr>
<th>EMPNAME</th>
<th>SKILLS</th>
<th>SALHIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING IS EMPLOYEE NAME</td>
<td>SKILLS</td>
<td>SALHIST</td>
</tr>
</tbody>
</table>

FIG 3-9. MODEL FOR DATA STRUCTURE OF FIGURE 3-3.
Encoding this distinction between basic items and more complex data structures as an axis in the model reflects an important distinction between data structuring techniques. An array of numbers is basically different from an array of decision table rows. On the other hand, most people would agree that an array of reals and an array of integers are both of the same genre.

Most common data structures impose an order on their constituent data elements. This need not always be so. The "Ordered" axis makes this distinction explicit. The mathematically-inspired set structure of Appendix A simply groups together a collection of data elements. Since the order of the individual condition names in a single row of the decision table of Figure 1-5 is unimportant, they could easily be considered a set. Such a set would be modeled as follows:

<table>
<thead>
<tr>
<th>Set</th>
<th>Homogeneous: YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic items: YES</td>
<td></td>
</tr>
<tr>
<td>Ordered: NO</td>
<td></td>
</tr>
<tr>
<td>Number: UNBOUNDED</td>
<td></td>
</tr>
<tr>
<td>Identification: NONE</td>
<td></td>
</tr>
</tbody>
</table>

The CONDITIONS set which is an element of the DTROW hierarchy from Figure 3-6 is shown in Figure 3-10. As an alternative to the array structure of Figure 3-2, SKILLS could also be modeled as an unordered set of ordered pairs describing a person's experience.

However, most aggregate data structures do impose some sort of order relationship among their elements. The array of Figures 3-1 and 3-5 defines an ordering of the five element instances in each array instance. The "Ordered" axis is answered YES whenever any sort of ordering is
FIG. 3-10. MODEL FOR A SET OF CONDITIONS FROM DECISION TABLE.
intended. Additional information may be specified with the YES answer to indicate the kind of ordering if it is not obvious. For instance, a two-dimensional matrix could specify either row or column ordering, or no ordering whatsoever.

Some programming languages require the user to declare the number of data elements instances which can be in an aggregate data structure. In the traditional array of Figure 3-1 the number of data element instances must exactly equal the range of the index. Alternatively, a programming language may allow the number of component data elements to vary but only up to some specified maximum (which is stated in the data declaration statement). Such a method is used by PL/I for arrays with the "varying" attribute. A few languages and several data base management systems provide aggregate data structures with varying numbers of components and for which the user need not specify any upper bound. For example, ALGOL 68 allows "multiples" (similar to arrays) of either fixed or unlimited (using "flex") size. Thus, while the first three axes are yes/no questions, the "Number" question allows three answers - FIXED, LIMITED, and UNBOUNDED. The VERS2 sequence of Figure 3-2 is a data structure of UNBOUNDED "Number." (Of course, some arbitrary limit on the number of components may be made by the implementation of these languages; however, the important point is that the programmer is not responsible for explicitly limiting the data structure.) This discussion of "Number" has centered on arrays; however, the same concept applies to all aggregates. There are languages and data base management systems which provide sets of fixed, limited, and unbounded number.

When the "Homogeneous" question is answered NO, the "Number" axis may be extended to specify a cardinality for each kind of data element. Figure 3-11 shows an alternative
FIG. 3-11. MODEL FOR ARRAY OF NONHOMOGENEOUS ELEMENTS.
view of part of the decision table. Each row of the decision table has previously been modeled as a hierarchy, one element of which was a set of conditions. In Figure 3-11 this same information is structured as an array. Unlike the traditional arrays considered previously, this DTROW array has two distinct kinds of elements. Each instance of DTROW has exactly one instance of an action and numerous conditions as its elements. In this case, it is very important that every row has exactly one action; multiple actions are not used by the decision table of Figure 1-5. If "Number" had only one answer, this restriction could not be modeled. Thus, this extension of "Number" allows some control over questions of data integrity.

The simple, contrived example shown in Figure 3-12 further demonstrates the need for this extension of the "Number" axis. Without the extension, the two data structures shown cannot be distinguished. Both are sets of exactly two elements where the elements may be integer and string basic items. However, the left-hand data structure is restricted so that every instance contains one integer and one string. To maintain data integrity, this restriction may be made part of the model. Thus, when "Homogeneous" is answered NO, "Number" may optionally be extended to specify a different count for each kind of element. This extension is not always needed, as demonstrated by the right-hand data structure in Figure 3-12.

"Identification", the last characteristic axis for aggregates, is also not a yes/no question. The purpose of "Identification" is to indicate how a specific constituent data element is named, pointed to, or labeled. "Identification" is a general term for this concept; the possible answers, as listed in Figure 3-4, are NUMBER, NAME, POINTER, and NONE. The traditional array data structure is indexed or
INSTANCES OF A
SET
HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: NO
NUMBER: INTEGER: FIXED, 1
STRING: FIXED, 1
IDENTIFICATION: NONE

INTESGER STRING

INSTANCES OF B
SET
HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: NO
NUMBER: FIXED, 2
IDENTIFICATION: NONE

INTESGER STRING

FIG. 3-12. USE OF "NUMBER" AXIS.
accessed by use of a NUMBER as an index. In the array of Figure 3-1, each element is labeled with a number from one to five. Users of this sort of array think of this number as identifying an element of the array. Similarly, the elements of an n-dimensional matrix are identified by an ordered n-tuple of numbers.

The concept of the "Identification" axis results from generalizing the ideas of an array or matrix index so that it can apply to other aggregates. A hierarchy structure, for instance DTROW shown in Figure 3-6, distinguishes its elements by NAME. For DTROW, these names are CONDITION VECTOR and ACTION. In contrast, a true mathematical set does not provide any way whatsoever of identifying its elements. In this case "Identification" is answered NONE because set elements are manipulated only with operations such as union and intersection; there is no way to access a specific, individual element of a set. The programming language PASCAL provides a data structure (called "class") which is very much like a set of limited "Number" except individual elements of it may be selected by use of a POINTER which is created whenever new elements are added.

The four possible answers to "Identification" do not exhaust all the possibilities. More detailed answers may be appropriate in some cases. For example, it might be useful to distinguish between different kinds of NAME identifiers: unique, system-wide names versus names which are valid only within a specified scope. Thus "Identification" may be adapted, within reason, to various applications. Appendix C illustrates various answers to this axis.

The discussion of the five characteristic questions for modeling aggregates is now complete. The next section uses some more examples to demonstrate how the model can be used.
3.2.2 Using the Aggregation Model

First, consider a few miscellaneous data structures. In Section 3.2.1, a set was modeled using the five questions. The particular data structuring technique selected was true to the mathematical definition of set in that "Number" was UNBOUNDED. However, a set of limited size may be desirable; such a data structure can be easily represented by the model, without introducing any new names, as follows:

<table>
<thead>
<tr>
<th>Bounded Homogeneous:</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Basic items:</td>
<td>YES</td>
</tr>
<tr>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td>Number:</td>
<td>LIMITED</td>
</tr>
<tr>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

(The name on the left is added for convenient reference only.) The elements of the bounded set must all be the same; however, a set consisting, say, of both real numbers and integers could be represented again by changing just one axis:

<table>
<thead>
<tr>
<th>Varied Homogeneous:</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Basic items:</td>
<td>YES</td>
</tr>
<tr>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td>Number:</td>
<td>LIMITED</td>
</tr>
<tr>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Finally, a set which could contain non-atomic elements of several different kinds but with the total number of elements still limited to some maximum is:

<table>
<thead>
<tr>
<th>Complex Homogeneous:</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td>Number:</td>
<td>LIMITED</td>
</tr>
<tr>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>
The point to be made is that numerous different kinds of sets can be modeled precisely without any confusion as to exactly what is implied by the term "set".

To further demonstrate the use of the five axes for aggregates, the aggregate data structures from Appendix A are all modeled using the questions; the results of this exercise are shown in Figure 3-13. Since the names of common data structures as they are used in contemporary systems are somewhat imprecise, there may be some debate as to the proper way of filling in Figure 3-13. For instance, several different versions of sets are sorted out above. Nevertheless, the answers appearing in Figure 3-13 faithfully represent the usual meanings of the terms "array", "matrix", "set", etc. (Examples of each of the aggregate data structures have been discussed in Section 2.)

Figure 3-13 allows some interesting observations. First, the five answers for set and repeating data structures are the same. This means that set and repeating are merely two different names for the same data structuring technique, one which allows an unordered, unbounded collection of like objects which are not distinguished by any special identification. (In this context, repeating structures which are accessed by specifying the value of some basic item are not included. Such data structures are in a different portion of Appendix A and are called "key files"). The only possible distinction between set and repeating might arise by answering the "Basic items" question YES for sets. Then there could be "sets" of real numbers and "repeating" structures of PEOPLE hierarchies. In any event, the model allows a clear definition of what is meant by these terms.

The next observation to be drawn from Figure 3-13 is that the row of answers for the relation structure is identical
<table>
<thead>
<tr>
<th></th>
<th>HOMOGENEOUS</th>
<th>BASIC ITEMS</th>
<th>ORDERED</th>
<th>NUMBER</th>
<th>IDENTIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
<tr>
<td>MATRIX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
<tr>
<td>SET</td>
<td>Y</td>
<td>N?</td>
<td>N</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
<tr>
<td>N-TUPLE</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
<tr>
<td>RELATION</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
<tr>
<td>HIERARCHY</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>FIXED</td>
<td>NAME</td>
</tr>
<tr>
<td>REPEATING</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
</tbody>
</table>

**Fig. 3-13. Model for all aggregates.**
to that of set (and repeating). This is indeed to be expected since a relation is typically defined in data base management systems (for instance [Codd 1971a, p. 68]) as a set of n-tuples. However, relations really need two levels of description, one for the set and one for the n-tuples. Figure 3-14 shows a two level pictorial data definition for a SKILLS relation which contains the same information as the SKILLS array of Figure 3-2. Further comments on the use of and need for multiple levels of definition are presented in Section 6.1.

Looking at Figure 3-13 in general, it is interesting to speculate on the total universe of data aggregates as modeled by the five questions. Clearly there are at least \(2^3 \cdot 3 \cdot 4 = 96\) different possible ways of filling the rows of Figure 3-13 at random (since there are three yes/no questions, three possible answers to "Number" when it is not extended for reasons of data integrity, and at least four possible answers to "Identification".) Thus, the model would seem to imply the existence of 96 distinguishable data structuring techniques for aggregates. Picking a random set of answers as an experiment, consider

<table>
<thead>
<tr>
<th></th>
<th>Homogeneous:</th>
<th>Basic items:</th>
<th>Ordered:</th>
<th>Number:</th>
<th>Identification:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous:</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>UNBOUNDED</td>
<td>NAME</td>
</tr>
</tbody>
</table>

which seems to indicate an aggregate of different type basic items selected by name. This data structure seems appropriate for a symbol table using hashing: the identification names are symbol names which are hashed to retrieve the current value of the symbol, which can be of different types (e.g., integer, real, address - all basic items).
FIG. 3-14. MODEL FOR SKILLS RELATION.
The preceding exercise suggests a further investigation of the possible aggregate data structures. Instead of just picking answers to the axes at random, it seems reasonable to pursue a systematic search of the total universe of aggregates. Figure 3-15 begins by listing, in a truth table-like form, all possible combinations of answers to the first three axes (those which have strict yes/no answers).

The eight rows of Figure 3-15 cover a wide range of possibilities. Rows 1, 2, 3, 7, and 8 duplicate initial parts of rows in Figure 3-13. In these cases, one of the aggregate structures from Appendix A has been listed in Figure 3-15 as a possible interpretation of the three answers. For instance, row 7 (Y, N, N) matches the repeating and set rows of the earlier figure. Starting from the row 7 answers shown, adding on possible answers to "Number" and "Identification" yields several data structures in addition to normal sets. Picking "Identification" NONE and varying "Number" results in structures like the bounded set considered at the start of this section (but with "Basic items" NO) and a fixed sized set in addition to the traditional unbounded set.

Not all of the rows of Figure 3-15 appeared in Figure 3-13. Rows 4, 5, and 6 thus do not represent any of the traditional aggregates. In these cases, Figure 3-15 shows a possible interpretation of what kind of aggregates might arise from such answers. Row 6, for example, suggests an amorphous, unordered collection of unlike things, all of which are basic items. Such a data structure would have no overall form or shape but simply provide an unsegregated box for keeping things in. One possible completion (i.e., values for the other two axes) was shown above titled "???".

Row 5 has completions like sets but restricted to contain only basic items (as was mentioned above as a possible
<table>
<thead>
<tr>
<th></th>
<th>HOMOGENEOUS</th>
<th>BASIC ITEMS</th>
<th>ORDERED</th>
<th>POSSIBLE INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>ARRAY, MATRIX</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N-TUPLE</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>SEQUENCE</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>ORDERED COLLECTION OF NONATOMIC ELEMENTS</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>SET OF ATOMIC ELEMENTS</td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>UNORDERED COLLECTION OF UNLIKE ATOMIC ELEMENTS</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>SET OR REPEATING</td>
</tr>
<tr>
<td>8</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>HIERARCHY</td>
</tr>
</tbody>
</table>

FIG. 3-15. TRUTH TABLE APPROACH TO DATA AGGREGATES.
way to distinguish between set and repeating structures). Data structures arising from row 4 seem quite unusual: ordered collections of any number of different kinds of not necessarily atomic elements. One example is:

Homogeneous: NO
Basic items: NO
Ordered: YES
Number: UNBOUNDED
Identification: NONE

which seems to represent a data structure which would be called a sequential file of unlike records in traditional terms, i.e., an ordered collection of unlike, nonatomic elements. The elements would be different types of records.

In some sense, the first row of Figure 3-15 is the most restricted while the last row is the most general. Any aggregates beginning from row 1 must have all like basic item elements ordered in some manner. Row 8, on the other hand, represents aggregates composed of different kinds of complex elements which are not ordered. Thus it seems reasonable to continue the exploration of the universe of aggregates by considering possible completions of rows 1 and 8.

Row 1 leads, with a few exceptions, to rather ordinary data structures (as is to be expected since it represents the most restricted answers to three axes). Variations of "Number" yield fixed, varying, and unbounded arrays, at least when the answer to "Identification" is NUMBER. (The identifier is a single number for arrays and an n-tuple for matrices.) If "Identification" is NONE, the result is a sequence restricted to basic items, i.e., an ordered collection of one kind of basic items lacking any identifying
marks. However, if "Identification" is NAME, an interesting, practical data structure results, as follows:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Homogeneous:</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic items:</td>
<td>YES, equivalence</td>
</tr>
<tr>
<td></td>
<td>Ordered:</td>
<td>YES, alphabetically by name</td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>UNBOUNDED</td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NAME, symbol name</td>
</tr>
</tbody>
</table>

Such a data structure is commonly used to store a compiler's symbol table. The contents are the equivalences or values of the named symbols. The symbol table is ordered so that it can be printed out as a cross-reference table. No common programming language makes this data structure available to its users.

Row 8 of Figure 3-15 leads down some unexplored paths to some interesting and some rather improbable data structures. They will be considered by fixing an answer to "Identification" and then considering the "Number" axis. Examining first the collection of data structures described by:

| Homogeneous: | NO |
| Basic items: | NO |
| Ordered:     | NO |
| Number:      | FIXED, LIMITED, or UNBOUNDED |
| Identification: | NAME |

we see, among them, the familiar hierarchy: a fixed sized collection of named components. The other two answers to "Number" (LIMITED and UNBOUNDED) are similar data structures. Both consist of a varying number of components which can be selected only by name; the components may be numerous kinds of nonatomic things. Such organizations are a little far fetched; they might be usable as generalized symbol tables to
store complex and different kinds of information about symbol names. Both these structures are really quite similar to the conventional hierarchy except that they provide a variable number of named components. The reader is cautioned at this point to recall that the model considers only the structural, time-invariant aspects of data structures. Thus the above set of answers should be viewed as describing a series of stop-time snapshots of the varying data structures just discussed.

When "Identification" is NUMBER, rather similar data structures result but with numbers replacing names in the above discussion. The components of the data structures are numbered but no ordering is implied. An example of a numeric identifier which might be used in this manner is the social security number. This identifier covers too wide a range to be used like an index and is not significant as an ordering.

Several programming languages, including PASCAL, provide data structures which may be modeled similarly to the following:

| PASCAL Class | Homogeneous: NO | Basic items: NO | Ordered: NO | Number: FIXED | Identification: POINTER |

In effect, a bunch of complex data structures are grouped together in a way so that individual elements can only be selected by a pointer which is created when the element is first added to the aggregate. The PASCAL "class" works in exactly this way, but limits the size of the class to a pre-defined number. Uses for similar data structures with other answers to "Number" are at least conceivable.
Finally, we have left to consider the case of "Identification" NONE. Such data structures collect together an unhomogeneous collection of various kinds of complex elements in a way so that the individual elements cannot be separately identified. Such data structures are similar to sets but with unlike elements. Practical uses for such data structures seem hard to find. Even if they are treated like normal sets, the semantics of their use become hard to define. For instance, what does it mean to intersect two pseudo-sets, one consisting of hierarchies and repeating structures and the other containing arrays and matrices? Interestingly, the SETL programming language does allow such structures. SETL sets are modeled:

<table>
<thead>
<tr>
<th>SETL</th>
<th>Homogeneous: NO, sets and n-tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Basic items: NO</td>
</tr>
<tr>
<td></td>
<td>Ordered: NO</td>
</tr>
<tr>
<td></td>
<td>Number: UNBOUNDED</td>
</tr>
<tr>
<td></td>
<td>Identification: NONE</td>
</tr>
</tbody>
</table>

Thus this data structure is a collection of any number of components which are themselves either sets or n-tuples. And, of course, the components of these components may also be either sets or n-tuples until atomic elements are reached. SETL, in addition to other ways of manipulating sets, does provide set union and intersection operators, but it is not clear how they are defined in the more esoteric cases.

This section has demonstrated at some length how the aggregate model can be used both to understand existing data structures and to explore new ones. Next, the development of the data structure model is continued with consideration of the association structures.
3.3 Association Model

Many different kinds of associations have been used by programming languages and database management systems; these have been surveyed in Section 2. Most associations are based on the idea of a pointer, "ref," or link between other data elements; the details vary greatly from system to system. This variation makes it difficult to state a set of modeling questions in an unambiguous manner. For this reason, this section first develops a generalized notion of association and then expresses the axis questions in terms of the generalized association.

An association is a pairing or binary relation between data aggregates and basic items. Associations are used to interconnect or tie together aggregates and/or basic items to add additional structure to the data. An association data definition specifies the data definitions of the aggregates and basic items to be related and the details of the association between them. An association instance associates together an appropriate number of data aggregate and basic item instances in the manner described by its data definition. Outline forms of two pictorial data definitions are shown in Figure 3-16.

For convenience, the two ends of the association are called the A-end and B-end. Neither end is superior or subordinate to the other; associations are not directed. The most common form of association has one kind of data aggregate or basic item at each of its ends; this type of association is shown at the bottom of Figure 3-16. More complex associations pair up two groups, each group containing several different kinds of data aggregates and/or basic items. This sort of association is defined in the way shown on the top of Figure 3-16. Figure 3-17 shows a simple example association.
A1
A2
Ai

NAME

B1
B2
Bj

GENERAL ASSOCIATION

NAME

B

A

SIMPLE ASSOCIATION

A, B: DATA DEFINITIONS OF DATA
AGGREGATES/BASIC ITEMS

FIG. 3-16. ASSOCIATION DATA DEFINITION.
FIG. 3-17. EXAMPLE ASSOCIATION.
The association OWNS relates cars and their owners. (This example, and several which follow, do not show a complete data definition for the ends. This omission allows attention to be concentrated on the details of the association.)

This concept of association bears some similarity to the "group relation" of [CODASYL 1971a, pp. 108-120] which was introduced for use in comparing data base management systems. The major difference is that the group relation distinguishes the ends; the data elements at one end are called "parent groups" and are superior to those at the other end which are called "dependent groups."

This thesis claims that the nondirected, nondistinguished-ends association described above is a more suitable basis for modeling the structural aspects of data structures. When considering structure alone, it seems clear that any sort of relation conceptually relates both its ends together in the same manner. An address pointer from A to B, for instance, certainly associates B with A as well as relating A to B. Likewise, although some associations such as the owner-member structure of Appendix A imply some sort of hierarchy, all associations do not. Thus [CODASYL 1971a, p. 108] mentions that the group relation is an alternative to the "hierarchic group." Finally, [CODASYL 1971a, p. 108] itself presents an example group relation between PERSON and SKILL in a way which is not at all hierarchic. (This example is similar to the table form of the association discussed immediately below.)

There are many ways to represent interrelations between two sets of objects. One common method is to use a "truth table" or two-dimensional table to represent which pairs of objects are related. This suggests an alternative pictorial view of association instance as defined above. Figure 3-18 redraws the three instances of OWNS from Figure 3-17 in this
FIG. 3-18. TABLE FORM OF ASSOCIATION INSTANCES FROM FIGURE 3-17.
The first axis, "Cardinality," describes how many data elements are related by each instance of the association. Figure 3-20 demonstrates the possible answers using the OWNS association introduced in Figure 3-17. In the OWNS association partially defined at the top of Figure 3-20, each person is associated with exactly one car, and vice versa; this version of the association has "Cardinality" 1-1. In the case of the OWNS association, other cardinalities are also reasonable. The next two associations shown in the figure are both "Cardinality" 1-N. The first associates many cars with a single person; the second relates many people to a single car. The final version of OWNS combines both the previous ones into a "Cardinality" N-M association which allows several cars and people to be interrelated. Note that the answer to "Cardinality" is given at the ends of the line representing the association in the pictorial data definitions.

Figure 3-21 offers a more complicated example of the use of the "Cardinality" axis. This example is drawn from the scheduling data base of [Frank and Sibley 1973] and contains two associations. (This data base is described in detail in Section 1.3 of this thesis.) This example shows how associations can be combined to form sophisticated data structures. Any instance of SKILL is associated with both the people who
1. WHAT IS THE CARDINALITY OF THE ASSOCIATION?
   HOW MANY INSTANCES OF A-END DATA DEFINITIONS ARE
   ASSOCIATED WITH HOW MANY INSTANCES OF B-END DATA
   DEFINITIONS IN ONE INSTANCE OF THE ASSOCIATION?

   CARDINALITY: 1-1, 1-N, N-M

2. HOW MANY KINDS OF DATA AGGREGATES AND BASIC ITEMS MAY
   OCCUR AT A-END (AT B-END)?
   HOW MANY DATA DEFINITIONS MAY A-END (B-END) INSTANCES BE
   DRAWN FROM?

   KINDS OF ENDS: 1, 2, ...
   (1, 2, ...)

3. MAY THE ASSOCIATION FORM A LOOP?
   MAY A-END AND B-END INSTANCES BE FROM THE SAME GROUP
   OF DATA DEFINITIONS?

   LOOP: YES, NO

4. IS THE ASSOCIATION COMPLETE AT A-END (AT B-END)?
   IS EVERY A-END (B-END) INSTANCE PART OF SOME INSTANCE
   OF THE ASSOCIATION (AS OPPOSED TO BEING UNRELATED)?

   COMPLETE: YES, NO

   (YES, NO)

   WHEN A DATA AGGREGATE OR BASIC ITEM IS AN END OF MORE THAN ONE ASSOCIATION, THE
   FOLLOWING QUESTION MAY BE ANSWERED FOR ANY SET OF TWO OR MORE ASSOCIATIONS
   CONNECTED TO THE END.

   *5. ARE THE ASSOCIATIONS EXCLUSIVE FOR THIS END?
      MUST EACH INSTANCE OF THE END BE PART OF EXACTLY ONE
      ASSOCIATION (AS OPPOSED TO BEING IN 0, 2, OR MORE)?

      EXCLUSIVE: YES, NO

      * DATA INTEGRITY

FIG. 3-19. MODEL FOR ASSOCIATIONS.
CARDINALITY: 1-1  EVERYONE OWNS EXACTLY ONE CAR

CARDINALITY: 1-N  EACH PERSON MAY OWN SEVERAL CARS

CARDINALITY: 1-N  MULTIPLE OWNERS OF EACH CAR

CARDINALITY: N-M  MULTIPLE OWNERS OF MULTIPLE CARS

FIG. 3-20. USE OF "CARDINALITY" AXIS.
DEFINITION

FIG. 3-21. TWO "CARDINALITY" N-M ASSOCIATIONS.
have the skill and the machines which require it. Both associations must be "Cardinality" N-M if the data structure is to be completely general and the restriction that there be only one instance of the SKILL aggregate for each skill is made. For instance, a single person can have several skills and each skill may be possessed by more than one person.

"Cardinality" can also be characterized in terms of the tabular association format which was introduced in Figure 3-18. For "Cardinality" 1-1, each row and column of the table may contain at most one X. The top association instance in Figure 3-18 is of this form. For "Cardinality" 1-N, either the rows or columns are restricted to contain at most one X. An example of this type of association instance is shown in the middle of Figure 3-18 where it is assumed that a person may own several cars. Since each car, however, is assumed to have one owner, each column of the middle instance contains no more than one X. Finally, for "Cardinality" N-M, there are no restrictions; the Xs may appear anywhere in the table. The bottom association instance in Figure 3-18 is of this form.

[Frank and Sibley 1973, pp. 5-11] has independently noted two of the possible "Cardinality" answers. The reference, while developing the example scheduling data base described in Section 1.3, distinguishes between "1 to N relationships" and "N to M relationships" among real world data. This distinction is vital to a proper understanding of the CODASYL Data Base Task Group "set" (see Section 3.3.2). [Whitney 1974] notes all three "Cardinality" answers, each as a different type of "association," and shows how to implement them in a relational data management system.
The next axis, "Kinds of ends," describes how many kinds of data elements may appear at the A-end and B-end of the association. The top data definition outline in Figure 3-16 shows the form of an association with more than one kind of data aggregate or basic item at each of its ends. [CODASYL 1971a] also allows "group relations" to have more than one kind of end. The only example of such a group relation provided by this reference has been redrawn in Figure 3-22 in the form suggested by Figure 3-16. This HAS SKILL association connects people with two different kinds of skills. USEFUL SKILL and USELESS SKILL could have completely different data definitions. Thus, "Kinds of ends" would be answered 1,2. In pictorial data definitions, the answer to this question is shown by enclosing the A-end or B-end data definitions inside dotted boxes if "Kinds of ends" is not 1. Similarly, the HAS SKILL association just considered could be summarized without a picture as follows:

Cardinality: 1-N  
Kinds of ends: 1,2  
Loop: NO  
Complete: NO,NO

"Kinds of ends" would be answered 1,1 for all the versions of the OWNS association considered in Figure 3-20. Figure 3-23 shows an association with "Kinds of ends" 2,2. The BUDGET association interrelates people and organizations with their supplies and salaries. Because it is "Cardinality" 1-N, each instance links either a single person or one organization with a collection of supplies and salaries. The association shown in Figure 3-23 is a basically different structure from the four similar "Kinds of ends" 1,1 associations shown in Figure 3-24. The former structure may be
FIG. 3-22. "KINDS OF ENDS" 1,2 ASSOCIATION.
DEFINITION

FIG. 3-23. "KINDS OF ENDS" 2,2 ASSOCIATION.

- 170 -
FIG. 3-24. ALTERNATIVE TO SINGLE ASSOCIATION OF FIGURE 3-23.
desirable for reasons of simplicity, generality, or efficiency.

The third axis for associations is "Loop." This question, which takes a simple YES or NO answer, determines if the A-end and B-end of the association may be the same data definition. Pictorially, if "Loop" is answered YES, the association forms a loop from a definition back to it. Such an association is shown in Figure 3-25; note that the answer to "Loop" is implied by the way the relation is drawn. The traditional name for the OFFSPRING association is a tree structure. Another traditional structure, the linked list, can also be easily modeled with "Loop" answered YES, as shown in Figure 3-26. The associations considered previously in Figures 3-20 through 3-23 all assumed "Loop" NO.

The example uses of "Loop" considered so far assumed "Kind of ends" was always answered 1,1. Figure 3-27 illustrates another version of the OFFSPRING association with "Kind of ends" of 2,2. In this case, MAN and WOMAN would both be defined by separate data definitions; each instance of the association associates a man or woman with N descendents. Thus, a particular instance of OFFSPRING may associate a man and a woman, two men, two women, or numerous other combinations of three or more people. In summary, when "Loop" is answered YES, it means that the association connects an instance of one of a group of definitions at its A-end with another instance from the same group of definitions at its B-end.

At this point it is timely to consider with a little more precision exactly what an instance of an association is. Some specific rules have been used up to now in drawing and counting instances in the figures; these rules will now be made explicit.
FIG. 3-25. TREE-LIKE ASSOCIATION WITH "LOOP" YES.
FIG. 3-26. LIST LIKE ASSOCIATION WITH "LOOP" YES.
FIG. 3-27. "LOOP" YES ASSOCIATION WITH "KINDS OF ENDS" 2, 2.
Figure 3-28 illustrates various instances of the familiar ONE-TO-ONE association between people and cars. The basic rule is: an association instance is a connected collection of end instances with the number of end instances determined by the "Cardinality" axis answer. That the number of instances must match "Cardinality" is straightforward. The need for the connected restriction is illustrated in Figure 3-28. Without this restriction, there would be no way to tell if the lower right hand collection of end instances were one or two association instances. (This restriction is necessary because "Cardinality" N-M does not determine an exact number of end instances.)

Figure 3-29 illustrates the rule for instances applied to associations with "Loop" answered YES. This figure uses the same tree-like and list-like data structures shown in Figures 3-25 and 3-26. Each individual instance is enclosed in dashed lines. In this case, a single end instance may participate in two instances of an association. For example, in the top right of Figure 3-29, Sue is a B-end of an OFFSPRING association whose A-end is Bob. Sue is also the A-end of another instance which relates her to Jim.

A similar overlap between end instances can also occur in instances of a data structure with two or more associations. For example, in Figure 3-21, the instance SKILL1 is an end of both a NEEDS SKILL association and a HAS SKILL association. SKILL3 is similarly in two association instances, while SKILL2 and SKILL4 are both in only one. The "Exclusive" axis (see below) distinguishes some of these different cases.

The remaining two questions of the association model are concerned only with data integrity. If data integrity is
FIG. 3-28. WHAT'S AN ASSOCIATION INSTANCE?
FIG. 3-29. ASSOCIATION INSTANCES WHEN "LOOP" IS YES.
being considered, these questions should be answered; otherwise, they may be ignored.

The "Complete" axis is used to specify if the ends of an association can exist independently without taking part in any association instance. Its use is easily illustrated with the same OWNS association between cars and people. Figure 3-30 shows two versions of OWNS with different "Complete" answers. The values of "Complete" are shown at each end of the association prefixed by a "C=". In the association shown at the top of the figure, instances of PERSON which are not related to any CAR as well as instances of CAR without an owner may both exist. Depending on the use of the data base, this may or may not be desirable. In the situation where all cars in the data base must be owned by a person, the model shown at the bottom of Figure 3-30 is appropriate. In this case, since "Complete" is answered YES for the CAR end of OWNS, every CAR instance must be part of some instance of the association. If the data base contained only car owning people, "Complete" would be answered YES for the other end of the association as well. For brevity, the "Complete" answer will often be shown only if it is YES.

The final axis, "Exclusive," is used in a somewhat different manner to record data integrity considerations. Whereas "Complete," when answered YES, prevents end instances from becoming "unconnected," "Exclusive," when answered YES, prevents the ends from being part of too many associations. The previous axes for associations all apply to the association itself. "Exclusive," however, is concerned with data definitions which are ends of two or more associations. For example, Figure 3-21 showed two associations, each of which had as one of their ends the SKILL data
People and cars exist independently.

Cars must belong to someone.

Fig. 3-30. Use of "complete" for data integrity.
definition. In this example, "Exclusive" would be answered NO for SKILL with regard to NEEDS SKILL and HAS SKILL since a particular skill may be both possessed by people and required by machines.

The example shown in Figure 3-31 shows how "Exclusive" may be used to restrict associations for reasons of data integrity. For the two associations shown, it is clear that a person cannot be both a current and former employee. Thus, for PERSON the "Exclusive" question is answered YES with regard to the CURRENT EMPLOYEE and FORMER EMPLOYEE associations. This answer is indicated pictorially by drawing a dashed loop around the two associations and showing the YES answer prefixed by "E=" inside the loop. The instances shown on the right of Figure 3-31 are invalid because, in each case, an instance of PERSON participates nonexclusively in more than one of the associations.

As indicated in Figure 3-19 where the entire association model is defined, "Exclusive" may be answered for any pair of two or more associations with a common end data definition. Thus, "Exclusive" may also indicate that each end instance may be in only one out of three, four, or more associations. It is also reasonable to allow some subset of the associations sharing a common end to be exclusive while other associations are not so restricted at the end. Examples of some of these more complex uses of "Exclusive" will be given in the next section. As with "Complete," "Exclusive" answers of NO will often be omitted.

The first axis introduced, "Cardinality," can also be used for data integrity. Figure 3-32 illustrates how interrelations between the "Cardinality" of different associations
FIG. 3-31. USE OF "EXCLUSIVE" AXIS.
FIG. 3-32. "CARDINALITY"Used FOR DATA INTEGRITY.
can be modeled. The data structure at the top of the figure uses two "Cardinality" 1-N associations; however, the drawing uses X and Y in place of N. The added notation "Y·X" means that any ORGANIZATIONs associated with the same BUDGET (Y) must also be associated with the same VICE PRESIDENT (X). Thus, the data structure shown assumes that organizations may share funds only if they are responsible to the same vice president. The middle of Figure 3-32 depicts the kinds of instances desired. The lower section of the figure shows the sort of instances which would be allowed if no data integrity restrictions were added to the data definition.

This discussion has introduced the section of the model which describes associations. Both structural and data integrity aspects are covered by the five axes.

3.3.2 Using the Association Model

This section further describes the association model by showing how it can be used to model some other common data structures and also by comparing it to another modeling scheme.

Data Structure Diagrams [Bachman 1969; Eriksen 1974] were proposed to model the association-like structure provided by the early data base management system, IDS. (Data Structure Diagrams are described in detail in Section 5 of this thesis.) These structures are similar to the 'group relation of [CODASYL 1971a] mentioned above; they are directed, 1-N relations. Data Structure Diagrams provide a concise way of representing a number of different variations of this sort of association.
The first use of the association model to be considered here is shown in Figure 3-33. This figure displays all the possible Data Structure Diagrams for a single association and expresses each possibility in terms of the model developed in the prior section. Figure 3-33 also suggests the corresponding structure in the currently popular data base proposal of the CODASYL Data Base Task group [CODASYL 1971].

Consideration of Figure 3-33 suggests that the association model of this thesis can easily model Data Structure Diagrams. For example, the basic Data Structure Diagram shown at the top of the figure can be represented with the model as follows:

Data Structure Diagram 1
Cardinality: 1-N
Kinds of ends: 1,1
Loop: NO

("Complete" and "Exclusive" answers are omitted since Data Structure Diagrams do not assume any of the data integrity restrictions.) Note that the model makes explicit the assumption that exactly one instance of the A-end will be related to an arbitrary number of B-end instances. The other Data Structure Diagrams in Figure 3-33 can also be easily modeled by changing the answers to the "Kinds of ends" and "Loop" axes as shown in the figure.

Figure 3-34 continues the modeling of Data Structure Diagrams, considering all possible diagrams of two associations. (Three of the forms are given the names shown by [Eriksen 1974].) These data structures can be easily represented by the model. In each case, two "Cardinality" 1-N associations are used.
FIG. 3-33. DATA STRUCTURE DIAGRAMS MODELED.
"COMPOUND NETWORK"

TWO SETS WITH SAME MEMBER

TWO SET WITH SAME OWNER

TWO SETS WITH SAME OWNER AND SAME MEMBER

TWO SETS, MEMBER OF FIRST IS OWNER OF SECOND

FIG. 3-34. MORE DATA STRUCTURE DIAGRAMS.
This exercise has shown that the association model can be used to depict the data structuring techniques which have been formalized as Data Structure Diagrams. It also provides an example of how the model makes clear the characteristics of a data structure without relying on a name.

The CODASYL Data Base Task Group proposal [CODASYL 1971] suggests a "set" structure which was included for comparison with the Data Structure Diagrams just considered. The association model can also be used to gain insight into this data structure. The DBTG "set" is rather more complex than most people's idea of set. In the catalog of data structures in Appendix A, it appears both in the owner-member structure and linked list rows. Figure 3-35 describes this rather complex structure with the model. As can be seen, more than a single association is necessary to faithfully model it. The MEMBERS association relates one owner instance with any number of instances of a group of members. This association is modeled as follows:

CODASYL DBTG          Cardinality: 1-N
Set MEMBER              Kinds of ends: 1,X X>1
                        Loop: NO

Then each member instance is related to the "next" (and thus also the "prior") member instance by another association which is modeled:

CODASYL DBTG          Cardinality: 1-1
Set NEXT               Kinds of ends: X,X X>1
                        Loop: YES
FIG. 3-35. CODASYL DBTG SET MODELED.
The ordering of member instances is defined by the "order is" clause in the DBTG proposal; it could be specified in the association model as an "is" comment. The DBTG proposal also includes data security and search information for sets; such information is not covered by the association model, but some aspects are included in the file model (see Section 3.4.2).

As shown in Figure 3-35, the DBTG set provides only a "Cardinality" 1-N relation between its owner and members. Since "Cardinality" N-M associations seem to be a useful representation of many real world situations, the DBTG proposal suggests a way of implementing them in terms of its "sets." This method is modeled in Figure 3-36 where it is applied to the HAS SKILL association considered earlier in Figure 3-21. As shown, two "Cardinality" 1-N associations and an additional end data element are used. (Note that this form is very similar to Data Structure Diagram 5 from Figure 3-34.) Each instance of the additional LINK data element is used to pair one person with one skill. The LINKs which are associated with a PERSON by the PERSON'S SKILL association are in turn related to the person's skills by HELD BY. This setup allows the full generality of a "Cardinality" N-M association: any N instances of SKILL may be related to any M instances of PERSON, and vice versa. (The number of special links required to do so equals the number of Xs which would be required if the original association were expressed in tabular form as in Figure 3-18.)

Figure 3-36 also illustrates the use of the data integrity questions "Complete" and "Exclusive." The LINK ends of both the HELD BY and PERSON'S SKILL associations have "Complete" answered YES. Every LINK instance must be part of both these associations if the suggested solution is to work
"CARDINALITY" N-M ASSOCIATION

IMPLEMENTED AS TWO "CARDINALITY" 1-N ASSOCIATIONS

FIG. 3-36. CODASYL DBTG SOLUTION TO "CARDINALITY" N-M LACK.
properly. (This implies "Exclusive" NO for the LINK end.) The other ends of each association have "Complete" answered NO because some people may have no skills and some skill may be possessed by no one. The "automatic" versus "manual" and "mandatory" versus "optional" distinctions of the DBTG set are related to the "Complete" question. The proposal also assumes "Exclusive" will always be answered NO.

The CODASYL DBTG set and one of the data structures consisting of multiple sets are thus shown to consist of different parts and each part can be understood with the association model. Although these data structures become rather complex, they are useful for representing many real world situations in a "network" style data base management system. [Curtice 1974] provides several illuminating and practical examples of uses for the more esoteric features of sets. The modeling exercises carried out above supply some much needed insight into why sets work the way they do.

It is also enlightening to further examine the "group relation" suggested by the CODASYL Feature Analysis work [CODASYL 1971a]. As mentioned in Section 3.3, the "group relation" associates a collection of "parent groups" with a collection of "dependent groups." (Like both the CODASYL DBTG set and Data Structure Diagrams, group relations are directed.) It seems fair to say that [CODASYL 1971a] is just a little vague about exactly how general "group relations" may be. Figure 3-37 models what is probably the most general structure intended. As can be seen, the group relation is considerably more complex than either the DBTG set or Data Structure Diagrams.

As mentioned in Section 3.3.1, although the definition of group relation clearly allows arbitrary answers to "Kinds
CARDINALITY: N-M OR 1-N OR 1-1
KINDS OF ENDS: \(X \geq 1, Y \geq 1\)
LOOP: YES OR NO
COMPLETE: ?
EXCLUSIVE: ?

FIG. 3-37. CODASYL FEATURE ANALYSIS GROUP
RELATION MODELED.
of ends," the most complex example provided is a simple "Kinds of ends" answer of 1,2 (similar to Data Structure Diagram 2 in Figure 3-33). Similar problems arise in trying to determine the proper answer to "Cardinality." In view of the definition of instance given [CODASYL 1971a, p. 112] and the example shown in that reference as Figure 2-20, "non-hierarchic group relations have "Cardinality" N-M. Likewise, "hierarchic group relations assume "Cardinality" 1-N and "non-repeating" group relations probably are intended to be "Cardinality 1-1. The lesson to be learned is that names are hard to define with English; the model suggested here leaves much less subject to the reader's interpretation.

To further demonstrate this difference, an example using several associations will be modeled. This example originally appeared in [CODASYL 1971a, p. 138] as shown in Figure 3-38 (the format is similar to a number of intertwined Data Structure Diagrams). Figure 3-39 redraws Figure 3-38 in terms of the association model. Additionally, whereas all the other examples considered in this section have shown only the details of the associations, Figure 3-39 includes a complete data definition of the ends in terms of the aggregate model. Since details of the ends were not provided in [CODASYL 1971a], Figure 3-39 has been fleshed out using the organization data base of Figure 1-1. This example begins to show how the data structure model is actually used to describe practical data organizations.

Since Figure 3-39 is the first fully detailed example, it is worth considering in some detail in order to see just what the model can do. The details which the model faithfully represents are best appreciated by simply listing the most interesting ones:

1. Each person works for exactly one organization; hence, EMPLCYS has "Cardinality" 1-N.
FIG. 3-38. CODASYL FEATURE ANALYSIS DATA STRUCTURE.
FIG. 3-39. DATA STRUCTURE OF FIGURE 3-38 MODELED.
2. The data base contains data only about people who are employees; thus, the PERSON end of EMPLOYEES has "Complete" YES. But, an organization may have no employees at the moment, so the other end has "Complete" NO.

3. Similarly, an organization may have many projects, each project belongs to exactly one organization, and some organizations have no projects; thus, IS RESPONSIBLE FOR is "Cardinality" 1-N, and "Complete" NO,YES.

4. A person may work on any number of projects at once, and more than one person may be assigned to each project, so WORKS ON is "Cardinality" N-M.

5. Each project has exactly one leader, some people are not leaders, but no one can lead more than one project; thus, IS LEADER OF is "Cardinality" 1-1 and "Complete" NO,YES.

6. A leader cannot also work on any project; thus, WORKS ON and IS LEADER OF are exclusive at their PERSON end.

7. People may have no skills and some skills may be possessed by no one; hence HAS is "Complete" NO,NO.

All the other combinations of relations which share ends are "Exclusive" NO.

As this last example indicates, the data structure model can do quite a bit for data integrity. However, it
cannot go all the way. The primary purpose of the model is to characterize the structural aspects of data organization; some kinds of data integrity remain beyond its reach. An example of a data integrity restraint which cannot be handled is shown in Figure 3-40. The data structure considered is a portion of the one just modeled in Figure 3-39. The desired data integrity restraint could be expressed: the leader of a project must be an employee of the organization responsible for the project. The instances shown differ in exactly that way; BOB and JILL lead projects which belong to each other's organizations in the undesirable instances. No combination of the five association axes can model this kind of restraint. Further consideration of what the data structure model (and particularly the part for associations) cannot accomplish is postponed until Section 7.1.

3.4 File Model

Files are the final general classification of data structures from Appendix A. This section describes a model for files and, hence, completes the data structure model. The model will be used and more examples of its use will be presented in Section 4 where the model is adapted to top-down design.

Files provide the connection or linkage between information and its user. A file defines a way of selecting or picking particular instances of some specific part of a data structure; that part is known as the file's entry.

A file takes a data structure and adds additional structure to correlate the entry with information known to the user. The original data structure may consist of basic
FIG. 3-40. UNSOLVABLE DATA INTEGRITY RESTRICTION.
items, aggregates, and associations between them. The entry may be one or more aggregates or basic items.

A file data definition specifies data definitions for a particular data structure, including one or more entries, and rules for selecting particular entry instances. Thus, the underlying data structure of a file, including the entries, can be modeled using the axes developed in the preceding sections. The model to be developed in this section must describe the method of picking entry instances from a collection of instances of the data structure.

Outline forms for pictorial file data definitions are shown in Figure 3-41. As mentioned above, the sections of the model previously described are used to depict a data structure. The file structure is drawn as a circle with an arrow connecting it to each entry data definition. The characteristics of the file are presented inside the circle.

The model considers files on a logical plane without details of implementation. In this way, a reasonably general file can be modeled without resorting to the introduction of a particular style or type of files (in contrast to what was required to gain an understanding of associations). The very simple, yet general, concept of file described above is adequate for studying the file structures of common data base management systems and programming languages.

A file instance consists of any number of instances of its component data definitions and a particular set of rules for picking entry instances. An example file instance will be shown in the next section.
FILE NAME
FILE CHARACTERISTICS

ENTRY 1

ENTRY 2

FILE WITH TWO ENTRIES

FILE NAME
FILE CHARACTERISTICS

ENTRY

FILE WITH SINGLE ENTRY

FIG. 3-41. FILE DATA DEFINITION.
It must also be noted that while files provide the means or basis for access, they are still independent of particular accessing schemes or methods. For instance, a sequential ordering of data elements may exist independently from any arrangement for sequential access. Thus the model will consider the structural aspects of file organization.

3.4.1 Model for Files

Figure 3-42 presents the four axes of the file model in the same format which has been used previously (Figures 3-4 and 3-19). Each question will be discussed individually.

The first axis, "Selection," describes the method used to select a particular entry instance from the file. Files were distinguished according to this characteristic in Section 2 and in Appendix A. The four types of "indexing" files shown in Appendix A correspond to the four possible answers for "Selection." The two kinds of "key files" are represented by the answers SPECIAL KEY and BASIC ITEM KEY. "Current pointer files" are modeled with the answer CURRENTNESS and "sequential files" which have nothing but strict sequential structure have "Selection" answered NONE.

Figure 3-43 shows a "Selection" BASIC ITEM KEY file; the REGISTRATION file is based on the familiar OWNS association. This file correlates a key value with all CAR instances which have SERIAL NUMBER components with the same value. An instance of REGISTRATION is shown in Figure 3-44. This figure shows the particular key values which are associated with each entry instance. Like all instances, instances of files change from time to time. The REGISTRATION file may
1. WHAT KIND OF SELECTION IS USED?
   WHAT DOES THE USER SPECIFY TO PICK
   ONE OR MORE ENTRY INSTANCES?

   SELECT: NONE, SPECIAL KEY,
   BASIC ITEM KEY, CURRENTNESS

   THE FOLLOWING QUESTION IS
   ANSWERED ONLY WHEN "SELECTION" IS BASIC ITEM KEY

2. IS THE ENTRY UNIQUE?
   DOES EACH SELECTION SPECIFY A SINGLE,
   UNIQUE ENTRY INSTANCE (AS OPPOSED TO
   TWO OR MORE)?

   UNIQUE: YES, NO

3. IS THE FILE SEQUENTIAL?
   IS ANY KIND OF ORDERING WHATSOEVER
   IMPLIED AMONG THE ENTRY INSTANCES?

   SEQUENTIAL: YES, NO

4. HOW MANY KINDS OF ENTRIES?
   HOW MANY DIFFERENT DATA DEFINITIONS
   MAY ENTRY INSTANCES BE DRAWN FROM?

   KINDS OF ENTRIES 1, 2, ...

• DATA INTEGRITY

FIG. 3-42. MODEL FOR FILES.
FIG. 3-43. REGISTRATION FILE WITH "KEY" A BASIC ITEM.
FIG. 3-44 INSTANCE OF REGISTRATION FILE
have cars added and deleted; this would affect the particular correlation between serial numbers and cars shown in the figure. The model does not reflect these changes; it considers only the unchanging structural nature of the file.

The other answers to "Selection" are also quite straightforward. BASIC ITEM KEY files have the actual key value appearing in the entry instance. SPECIAL KEY files, on the other hand, use as a key value a quantity which probably has no particular meaning to the user. (An exception is when the special key is a "record number.") With "Selection" of CURRENTNESS, the user does not use keys directly. Instead, the system keeps track of the most recently selected entry instances and allows them to be selected at will. Many kinds and variations of current pointer files exist; with the file model, additional information can be used to specify the details of a particular scheme. Finally, some files may not provide selection of particular entry instances; they simply group together a collection of instances of some data structure, possibly for sequential access. Examples of these various answers to "Selection" will be used below and in Section 3.4.2.

The next axis provides some control over data integrity when "Selection" is answered BASIC ITEM KEY. The "Unique" axis records the fact that in some files there may be a unique entry instance corresponding to any key value, while in others a whole set of entry instances may be selected by a single key value. The REGISTRATION file considered in Figure 3-43 would presumably have "Unique" YES since serial numbers are intended to uniquely identify cars. Two cars with the same serial number should not be allowed in such a data base.
Another file, also based on OWNS, is shown in Figure 3-45; this file has "Unique" NO. The MANUFACTURE file associates any number of cars (entry instances) with a key value such as "74 Ford." This file also shows that a basic item key can consist of two or more basic items.

When "Unique" is answered YES, the correspondence between key values and entry instances can be viewed as a mathematical partial function: the file maps each element in its domain (key values) into at most one range element (entry instance). When "Unique" is NO, the correlation is not a mathematical function since one key value may correspond to more than one entry instance.

The other possible answers to "Selection" all imply a "Unique" answer of YES. Current pointers, for instance, always select a unique entry instance. In these cases, the "Unique" answer may be left blank or a YES filled in even though no further information is provided by such an answer.

The next axis, "Sequential," determines if there is any sort of ordering among the entry instances. This ordering may exist independently of the answer to "Selection." Figure 3-46 presents an alternative version on the REGISTRATION file organized in a sequential manner in addition to using basic item keys. This version adds a logical ordering of the entry instances according to the year of the car's manufacture. The details of the ordering are specified as additional information following the YES answer to "Sequential." It is clear that there are two sorts of orderings: increasing and decreasing. Also, a file may have both major and minor
FIG. 3-45. MANUFACTURE FILE WITH "UNIQUE" NO.
FIG. 3-46. REGISTRATION FILE WITH BOTH KEY AND SEQUENTIAL ORGANIZATION.
orderings. For example, the REGISTRATION file could be ordered first by YEAR and then by MAKE within each year. All information of this sort is specified as additional information. The "Sequential" axis distinguishes only the basic structural difference between files which have any sort of order and those which have none.

The files considered in Figures 3-43 through 3-46 all have a single kind of entry, i.e., the user of the file always selects an instance of the same data definition. This need not always be so; the "Kinds of entries" axis models this fact. Figure 3-47 displays a file with "Kinds of entries" answered 2. Note that the answer to this axis is implicitly shown in the pictorial data description: the answer equals the number of arrows between the file's circle and entry data definitions. The data structure of the SKILLS file in Figure 3-47 is the HAS SKILL association with "Kinds of ends" 1,2 from the previous section. The file correlates a useful or useless SKILCODE with the appropriate kinds of instances. From the entry instance, the associated person who HAS the SKILL can be found by "navigating" through the data structure.

In the previous example, the file's data structure contained an association which was very "compatible" with the file, i.e., the file was "Kinds of entries" 2 and the association was "Kinds of ends" 2 at its appropriate end. This need not always be the case. Figure 3-48 illustrates another file with two kinds of entries which, in this case, are not part of the same association. The data structure of this file was also considered in the previous section; the HAS SKILL and NEEDS SKILL associations connect people who possess
FIG. 3-47. SKILLS FILE WITH TWO KINDS OF ENTRIES.
FIG. 3-48. SCHEDULE FILE WITH TWO KINDS OF ENTRIES.
skills with machines which need the skills. The PERSON and MACHINE hierarchies use a SCHEDULES aggregate which would be defined separately, as shown in the lower corner of the figure. The SCHEDULE file connects a date with the people scheduled to work that date and with the machines scheduled to be used. Note that "Unique" is answered NO since many people probably work with many machines on every date. Thus any particular date is linked to both instances of PERSON and instances of MACHINE, as shown in Figure 3-49. Figure 3-49 includes only the PERSON and MACHINE instances (at the top and bottom of the drawing respectively) and does not show SKILL instances or either of the associations.

This SCHEDULE file (Figure 3-48) and its example instance (Figure 3-49) demonstrate two interesting aspects of files with "Selection" BASIC ITEM KEY. First, the basic item whose value the key matches may be a subelement of the actual file entry. In the SCHEDULE file, instances of either PERSON or MACHINES (the two entries) are selected according to the values of the WORK or USE subelements of the two hierarchy structures. Thus, the key matches a lower level component of the entry. A more exact way of representing this with the pictorial data definitions will be presented in Section 4 as part of the top-down method for describing data structures.

Second, any instance of the entry in a file may have multiple occurrences of a basic item on which a file is based. In the example SCHEDULE file, a particular instance of PERSON may be selected by any number of dates. In this case, "Unique" may still be answered YES or NO, depending on whether the various key values each occur in exactly one place or not.

The axes of the file model have now been discussed. Each of the examples used to illustrate the model has considered only a single file; however, typical uses may require
FIG. 3-49. PARTIAL INSTANCE OF FIGURE 3-48.
several files which connect a data structure to its users. Figure 3-50, which uses the same data structure as Figure 3-48, provides three separate files. In a generalized database management system, different users may use the stored information in different ways; thus, various sorts of connections between keys and entries may be provided. The data definitions of Figure 3-50 show a sequential PAYROLL file, and keyed SKILLS and EQUIPMENT files. Different users may wish to enter the data structure at any of the three points shown.

An important aspect of the file model is that it characterizes a file independently from the kind of data organization it contains. Using the classical term, the file model does not distinguish between files according to the type of "records," but rather according to the kind of correspondence between the entries and the user. In the following section, the model will be used to describe some file structures from existing data base management systems; such files usually are not distinguished from the data structure which they use.

3.4.2 Using the File Model

Perhaps the best way to appreciate the file model is to see the clarity it can lend to file structures of the CODASYL DBTG proposal [CODASYL 1971, pp. 125-148]. Numerous, optional intertwined files are proposed, all of which are relatively inseparable from the underlying "owner-member" data structure. (This data structure has been modeled and discussed in Section 3.3.2.) Figure 3-51 is a composite model of all these files. The files are simply named A, B, C, etc., in the figure; their characteristics will be discussed in the text.

First, each member "record" has a file automatically associated with it independently of its occurrence in the
FIG. 3-50. MULTIPLE FILES.
FIG. 3-51. CODASYL DBTG FILES.
The answer to "Selection" is determined by the "location mode" clause given with the record's definition. Location mode of "direct" corresponds to SPECIAL KEY, and "calc" to BASIC ITEM KEY. When "calc" is specified, the answer to "Unique" is given in the "duplicates allowed" clause. The location mode may also be "via" in which case the record is not an entry; instead, the record instance will be found by navigating down to it from its owner. (This implies the record will later be defined to be part of a set.) Finally, no sequential access is possible when a record stands alone. Thus, the A files of Figure 3-51 are the basis of a straightforward form of keyed access. (Although the DBTG "Data Definition Language" does not include a provision for noting it, the "Data Manipulation language" also makes available a current pointer for each kind of record.)

By virtue of taking part in a set structure, each member acquires another file, shown as file B1 in Figure 3-51. These files are more general than the A files just considered - they provide both sequential and random access. File B is modeled:

DBTG B1
Selection: BASIC ITEM KEY
File
Unique: YES or NO
Sequential: YES
Kinds of entries: 1
The "ascending" or "descending" clauses provide increasing and decreasing major and minor orderings based on components of the particular member record. The keyed aspects of the file are defined with the "search key" clause (which also allows user selection of alternative implementation techniques). The basic items used as keys in the B files may be the same or different from those used in the A files.

Each DBTG set also has two files associated with the entire set (as opposed to the A and B files which connect to the individual member records). The first of these, shown as the C file in Figure 3-51, is modeled:

DBTG C File
Selection: NONE
Unique: -
Sequential: YES
Kinds of entries: equals number of members

The multiple entry, sequential C file is controlled by the "order is" clause. The simplest form of "order is" specifies a chronological ordering. Other forms define various alternative major and minor orderings based upon the record names, the items used to order the B files, or the special keys which may be used with the A files. Thus the C file can have a little bit of everything thrown into its ordering criteria. In all cases, the C file is strictly sequential; no forms of indexing are supported.

The final file, D in Figure 3-51, is another implied "Selection" CURRENTNESS file. The Data Manipulation Language for COBOL provides a "current of set-name" qualifier that can select any member or owner instance. This D file is modeled as shown on the next page.
This current pointer is maintained as a side effect of other commands in the Data Manipulation Language.

It seems fair to say that this exercise with the file model provides a simple, easy to understand insight into the DBTG data organization. It at least supplies a clear distinction between the "set" structure itself and the files associated with it. Some of the finer nuances and many of the implementation details which the DBTG proposal includes are absent from the model; however, the model does provide a simple, yet definitive, description of the logical structure of this data organization.

The CODASYL Feature Analysis work also defines a generalized "file" [CODASYL 1971a, pp. 134-144]. The data structure shown in the previous section as Figure 3-38 was introduced by Feature Analysis as "a file with multiple (group) entry schemas." This so-called file will be modeled here and some questions raised about the Feature Analysis file concept. First, the distinctions between the term entry as used here and in CODASYL Feature Analysis should be noted. [CODASYL 1971a, p. 120] defines an entry as "a set of groups and group relations in which one and only one group, the entry-defining group, is not contained in or subordinate to any other group. The term entry used in this thesis is most similar to the Feature Analysis entry-defining group but without the restriction that it not be related or associated with other parts of the data structure. An entry, as used here, may be any part of the data structure which is selected by a file, regardless of which associations it participates in.
The above generalization of the entry (or entry-defining group) is obviously quite useful. Figures 3-43 and 3-45 show two files based on the same OWNs association. If OWNs were a directed group relation, either its CAR end or PERSON end would be subordinate to the other and hence not acceptable as a Feature Analysis entry-defining group. Thus only one of the files shown in these two figures would be permitted.

Figure 3-52 models the Feature Analysis example considered earlier in Figure 3-38; the figure contains four files, one for each of the Feature Analysis entry schemas. The exact characteristics of the files were not detailed in the original work; those shown in the figure are reasonable possibilities. Full data definitions for each of the aggregates (PERSON, etc.) have been omitted in Figure 3-52; they would be as shown in Figure 3-39. The four files shown each provide independent entry to the information; this is what the Feature Analysis multiple entry schema file means.

CODASYL Feature Analysis distinguishes associations between entries such as those in Figure 3-39 from other associations whose ends are not entries (in the terminology developed here). Associations of the kind in Figure 3-52 are called "inter-entry group relation schemas" and are considered part of the file itself. This distinction seems to be unnecessary and slightly confusing. Figures 3-52, 3-39, and the considerations within Section 3.3.2 show that such associations can be considered and modeled like any other association. In fact, the modeling described in Section 3.3.2 and shown in Figure 3-39 was motivated without consideration of the data structure's use in a file. These same axis answers apply equally well to the files modeled here; thus, the distinction seems unnecessary.
FIG. 3-52. CODASYL FEATURE ANALYSIS FILES.
The distinction is also confusing because entry schemas can themselves contain normal group relation schemas (in the case of "tree" or "plex" entry schemas). This makes it hard to determine when a particular group relation is inter-entry (and thus part of a file) and when it is part of an entry. In Figure 3-38, for example, two alternatives seem equally reasonable. The first interpretation, implied in the original text, sees this data structure as four group entry schemas. But it is equally reasonable to consider SKILL as a subcomponent of a tree entry composed of PERSON, SKILL, and the HAS relation. In this case, the data structure has only three entries. The whole thing could equally well be a single plex entry with ORGANIZATION UNIT at its head (assuming REPORTS TO is "non-hierarchic").

These problems arise because of a confusion between entries and files; the two separate concepts seem to overlap because of the way group relations can be used. The file model developed here avoids these problems by treating the file's characteristics separately from any modeling of its underlying data structure.

Both the examples of this section show the clarity of definition possible with the file model. The examples also demonstrate how the model can be used to gain insight into complex data organizations.

3.5 Summary

Section 3 has exhibited a model for common data structures in three categories: aggregates, associations, and files (as shown in Appendix A). Each section of the model consists of a group of questions or axes; one section of the model is presented in each of Figures 3-4, 3-19, and 3-42.
In these figures, the aspects of the model which apply only to data integrity are marked with an asterisk.

This model succeeds in describing data structuring techniques without using either new or old names for each particular data structure. Instead attention is concentrated on the basic structural differences between data structures: each question and its possible answers describe one axis along which data structures may vary.
4. **TOP-DOWN DATA DESCRIPTION**

In this section, a top-down design methodology for data structures will be proffered. This method is based upon the data structure model developed in the previous section. The use of the top-down method is investigated in Section 6.2, where it is applied to a software development data base.

This section introduces the major precepts of structured programming concentrating on the top-down approach. Then papers of Dijkstra, Mills, and Wirth, representing the genesis of structured programming, are examined to determine the essence of various approaches to top-down design. Next this survey is used to motivate a method for the top-down description of data structures. This method is then applied to some of the examples used throughout this thesis.

4.1 **Introduction to Structured Programming**

Structured programming has remained a somewhat nebulous term; the very fact that it has eluded definition has prompted papers [Denning 1974; Gries 1974; Karpinski 1974]. An up-to-the-minute definition which incorporates most aspects of what now passes as the structured approach to programming is offered by [Ledgard 1974].

This lack of concise definition notwithstanding, most people would agree that the fundamental purpose of structured programming is the better understanding of programs. Dijkstra, the founding father of structured programming, has described this goal as the "intellectual manageability" for programs [Dijkstra 1972a]. There are two basic areas
in which structured programming has suggested changes: control structures and program design.

Structured programming is best known for its advice that the GOTO be eliminated from the control structures offered by programming languages. Dijkstra began the attack on GOTO with a now famous letter in the Communications of the ACM [Dijkstra 1968]. This letter and several other papers posit the GOTO as the culprit in the creation of "rat's nests" of program flow; it is claimed that programs without GOTO are usually more readable and easier to understand. The argument that GOTO be eliminated is made more credible by several formal proofs that all possible programs can be written without it [Ashcroft and Manna 1972; Bohm and Jacopini 1966]. However, there are some cases where the GOTO is the most straightforward solution to program requirements. Most people now agree that certain restricted sorts of GOTO, perhaps disguised with names such as ESCAPE, LEAVE, or EXIT, are useful [Wulf 1972].

The control structure aspects of structured programming have little to offer in the way of motivation for a top-down data structure design method. However, there has been some attempt to draw an analogy between the use of GOTO in unstructured programs and the unrestricted use of pointers to create ad hoc data structures [Hoare 1973, p. 3; Shnederman and Scheuerman 1974, p. 566]. In both cases, the argument is that GOTO and pointers are used by the system (either compiler or data manipulation routines) to implement the programmer's requests and, thus, the programmer is better off not to directly use such powerful and unrestricted features.

The other main area embraced by structured programming is program design; this topic is apposite to the task at
hand. The existing approaches to structured program design offer the primary motivation for the top-down data structure method detailed in Section 4.3.

The program design methodology suggested by structured programming has been called "top-down design" [Kills 1971], "stepwise refinement" [Wirth 1971a], and "levels of abstraction" [Dijkstra 1972]. All of these approaches are similar in their general organization. A suitable, general definition of top-down design is: Design based on "levels" making use of "abstractions" which will be described in a different level; each level is a readable, understandable entity which can be considered in a stand-alone fashion. For the reader to whom level and abstraction are not sufficiently primitive terms, the following dictionary definitions are offered:

level: an extent, measure, or degree of achievement.

abstraction: an abstract or general idea or term; the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.

Thus, a level is the extent of a program design up to some point using general characteristics which will be defined at another level. In most top-down designs, the levels are named or numbered in a chronological order. In this way, an early stage of the design, say the ith level, may freely assume any number of abstractions which will be defined at the i + 1st or later levels.

This same conceptual regimen may be applied to data structures. The next section describes several methods of top-down design and Section 4.3 applies these ideas to data structures.
4.2 Contemporary Top-Down Program Design

This section surveys the works of three people who have devised and popularized the concepts of top-down design. The three, Dijkstra, Mills, and Wirth, will be considered separately first (in alphabetical order) and then a common summary will be attempted.

Edsger W. Dijkstra's works on structured programming include [Dijkstra 1970; Dijkstra 1972; Dijkstra 1972b; Dijkstra 1968a]. The major emphasis of this work is that a program's structure should be tied to a "convincing demonstration" of its correctness; thus Dijkstra also refers to this approach as a "constructive approach to the problem of program correctness." The goal of correct programs is seen as a mandate for readable programs since otherwise it will be quite hard to make the demonstration convincing. Dijkstra suggests two ways to make programs readable: simple control structures and abstraction.

Dijkstra sees abstraction as an application of the "golden principle divide and rule" [Dijkstra 1972, p. 28] and presents several different ways of using it. The best developed method is the "string of pearls" [Dijkstra 1972, pp. 50-63; Dijkstra 1970, pp. 87-88] which will be described here.

Each level of a "step-wise program composition" is described as a "machine" with a meaningful name and one or more "named algorithms." The components of such a machine are "instructions" and "variables" of certain "types." The algorithms are expressed in terms of the instructions and variables using a typical ALGOL-like programming language. An example machine [Dijkstra 1972, p. 51] is:
This machine is named COMPFIRST representing a design decision to compute all the required output then print it. It defines a single algorithm "draw" in terms of the abstractions "build," "print," and "image." The meaning or requirements placed upon these abstractions are not formalized as part of the machine. Apparently Dijkstra is content to let the abstractions' names imply a sufficiently general concept. The thought processes involved, however, put some definite demands on these abstractions. For instance, in this example, "build" must compute and save in its parameter "image" 1000 coordinate values for "print" to output. These requirements are presented in the text surrounding the machine, but this text is attempting to describe the top down method and it is not clear how much of such commentary information Dijkstra would include in a practical top-down design. In another style of top-down programming [Dijkstra 1972, pp. 26-39] he does include quoted English phrases within the text of a program. And in still another style [Dijkstra 1972, pp. 77-80] he uses long, meaningful names for abstractions (e.g., SET QUEEN ON SQUARE [0,h]).

The example machine shown above uses a number of abbreviations. For instance, when there is only one variable of a given type, the name of the type (e.g., image) is used as a variable name also. Dijkstra says of these conventions: "I do not yet know whether they are very wise or very foolish."
An individual level or machine contains any number of algorithms; a level is picked on the basis of a single design decision (encoded in the machine's name). Thus all the algorithms in one level are the result of a single decision. This is again an application of "divide and rule." At any given point in the design process there will be some abstractions (instructions and types) awaiting definition. The group of these which can be refined after making the next, individual design decision are collected together and defined in the next machine.

This procedure raises the question of how the next design decision is to be made. Dijkstra's advice is to pick, as the next abstraction to be defined, the one which is not affected by the other abstractions or the one which can be selected without "further commitments" [Dijkstra 1972, pp. 52-54].

In the example machine above, Dijkstra argues "the action 'build,' however, admits a further detailing all by itself;" i.e., the "build" abstraction does not depend on the future definition of either "print" or "image." The next machine, formalizing the design decision to clear the image before computing the coordinates, is thus:

```
CLEARFIRST
begin
build: {clear, setmarks};
instr clear(image), setmarks(image)
end
```

This machine defines the abstract instruction "build" of the first level (COMPFIRST) in terms of two new abstractions and the existing "image" abstraction. Dijkstra sums up his advice for selecting levels as follows: "Programming (or problem solving in general?) as the
judicious postponement of decisions and commitments!"
[Dijkstra 1972, p. 67].

Dijkstra considers one special kind of level or refinement which deals exclusively with data structures [Dijkstra 1970, p. 87]:

"In the refinement of an abstract program ... we observe the phenomenon of 'joint refinement.' For abstract data structures of a given type a certain representation is chosen in terms of new (perhaps still rather abstract) data structures. The immediate consequence of this design decision is that the abstract statements [instructions] operating upon the original abstract data structure have to be redefined in terms of algorithmic refinements operating upon the new data structure. Such a joint refinement of data structure and associated statements should be an isolated unit of the program text: it embodies the immediate consequences of an (independent) design decision ..."

This process of implementing abstractions with machines can be likened to stringing a necklace from pearls. Dijkstra eloquently introduces the idea as follows [Dijkstra 1972, p. 59]:

"One of the metaphors in which I find myself thinking about the program structure envisaged regards the program as a necklace, strung from individual pearls. We have described the program in terms of levels and each level contained 'refinements' of entities that were assumed available in higher levels. These refinements were either dynamic refinements (algorithms) or static refinements (data structures) to be understood by an appropriate machine. I use the term 'pearl' for such a machine, refinements included."

The levels or pearls are thus strung in a linear fashion to form a complete program. At any given stage of the design the pearls which have already been strung define the program in terms of the remaining abstractions.
Dijkstra admits the possibility that the designer may not make the proper decision at each level and that such an error may not be evident until later levels are considered. This event then causes a certain amount of "reprogramming" of the earlier levels. (An example of such a backup in an alternative top-down design formalism appears in [Dijkstra 1972, pp. 34-36].) In terms of the string of pearls approach, one or more pearls must be unstrung and replaced with new ones representing the modified design decision.

Dijkstra also cautions against too loose usage of the process of abstraction [Dijkstra 1972b, p. 4.8]:

"But the fourth thing is probably the worst: apparently they [people trying to organize large scale design projects] do not know the difference between 'vague' and 'abstract' where it is the function of abstraction to create a level of discourse where one can be absolutely precise!"

He believes that a convincing correctness argument must be made in terms of precise, specific, well understood abstractions.

Dijkstra suggests the normal closed subroutine as the proper way to implement a program as a number of levels. The $i$th pearl defines subroutines which are used by one or more of the pearls above it. By keeping the individual pearls present in the final program, Dijkstra feels program modification becomes easier. He states: "The pearl, embodying the independent design decision or, as the case may be, an isolated aspect of the original problem statement, is meant to be the natural unit for such modification" [Dijkstra 1972, p. 60]. Thus program
modification is viewed as unstringing the necklace and reforming it adding new pearls where appropriate.

In summary, Dijkstra promulgates top-down design as a linear sequence of levels, each level formalized as a machine with abstract instructions and data types. Each machine may give definitions for as many abstractions as can be refined based upon a single design decision. All the levels are present in the final program which uses traditional subroutines to connect abstractions and their definitions.

Harlan D. Mills' work on program design [Mills 1971; Mills 1975] presents several variations to Dijkstra's top-down approach. Mills, of course, remains within the same general bounds (programs as top-down levels), but suggests a different view of what should be in a level and how levels are picked.

Mills visualizes a program design as a top-down tree. Each node of the tree, called a "segment" is an independent part of the program. Segments are named and consist of three kinds of information:

1. The actual algorithm expressed in the programming language being used and named abstractions,

2. A "functional specification" for each abstraction used, and

3. "Documentation" for the segment.

Expressing the levels in a programming language allows top-down program integration and testing to proceed along with the design process. Mills states: "In the structured
programming process, this design structure is carried out directly in code, which can be at least syntax checked, and possibly executed, with program stubs standing in for functional subspecifications" [Mills 1971, p. 43]. (For more details on the concept of "stubs" see [McHenry 1973]).

Each abstraction used in a segment will be defined in another segment at the next level of the tree structure. The name used for the abstraction becomes the name of a segment on the next lower level. At the original level, where the abstraction is first introduced, its functional specification defines the characteristics assumed of the abstraction. This specification treats the still-abstract segment-to-be as a functional "data transformer." Each segment is viewed as an operator which converts input data to output data; Mills states: "A function specification corresponds to the mathematical idea of a function" [Mills 1971, p. 50].

The role of the functional specification is to define the assumptions placed on the abstraction; i.e., to describe what the unwritten subsegment is supposed to do. In this writer's opinion, Mills intends the specification to be expressed either in English or in the functional formalism of [Mills 1975]. An example of a level defining an algorithm "g" is [Mills 1971, p. 53]:

\[
g \text{ expands to: } \begin{cases} \text{IF } p \text{ THEN } i \text{ ELSE } j \end{cases}
\]

Subspecifications (Level 2)
\[
p = \text{"Member name is in index"} \\
i = \text{"Update text pointer"} \\
j = \text{"Add name and text pointer to index"}
\]

The specification for g had already appeared at level 1. In the level 2 specifications, i and j can clearly be seen as data transformers (of the "text pointer" and
"index" respectively). Apparently, p is an identity data transformer (examining but not changing "index") which produces a useable result which can be tested by IF. Mills does not distinguish between these different types of abstractions.

The third component of each segment, its documentation, provides a "proof" that the segment properly implements its specification. Mills speaks of designing the tree structured program as an "expansion" process [Mills 1971, p. 42]:

"Each functional subspecification defined in an intermediate system represents only a mapping of initial data into final data for some segment of coding yet to be specified. The expansion process describes the means selected for this mapping, using possibly more detailed mappings to be similarly described later."

This author's graphic interpretation of this expansion is shown in Figure 4-1. The purpose of the documentation is to provide the "proofs" shown as dashed lines in the figure. The proofs are retained as the documentation that the subsegment correctly performs its specification.

Mills hedges slightly about the rigor of these proofs [Mills 1971, p. 51]:

"The specifications may be too complex to carry out a completely rigorous proof of correctness, but at the very least, there is on one page a logical description of a function which can be heuristically compared with the functional specification for that segment."

Thus it seems that what Mills wants is more in line with Dijkstra's "convincing demonstration."
FIG. 4-1. TOP-DOWN EXPANSION (MILLS).
Continuing to compare Mills' and Dijkstra's approaches, several differences are apparent. First, each of Dijkstra's levels is a single machine, possibly containing multiple algorithms which may share variables and types and use the same abstractions (instructions). For Mills, a level means a collection of tree nodes all of which are at the same level from the root. These nodes need not be related in any specific way; they are all independent refinements of the immediately preceding segments. For Dijkstra, one level's abstractions need not be defined in the very next level but may be postponed until any later pearl. Dijkstra's levels are intimately tied to the decision making process; Mills' levels group together a particular collection of abstractions (i.e., the abstractions used for the $i$th level of the design).

Another difference between the two approaches is the method of picking abstractions. Mills suggests that choices be made to define the interface between separate abstractions as quickly as possible. This allows the design of the abstractions to be carried on independently, perhaps by different people. Dijkstra specifically refutes this view [Dijkstra 1972, p. 62] which he expresses: "the well-known advice: if you are faced with two primitives ... decide immediately upon their interface ..."

Finally, whereas Dijkstra retains the separate levels in the eventual implementation of the program, Mills does not. Instead, Mills advocates a macro-like substitution facility to automatically insert a subsegment's code wherever its name has been used. He suggests an automatic library system for storing the current version of each segment (and perhaps stubs); the compiler would extract and insert the proper subsegments whenever a segment is compiled [Mills 1971, p. 46]. When representing top-down
designs solely on paper Mills simulates this effect with what he terms a "restatement." A restatement reexpresses the current state of the entire program's design by substituting all the current segments back into the top level; this results in a single, consolidated version of the program expressed in terms of the (currently) lowest level abstractions. A similar restatement facility is included in the top-down data structure design method described below.

In summary, Mills designs a program as a top-down tree of relatively independent algorithms. Each algorithm is described in a programming language and makes use of numerous abstractions which are described in English or mathematical notation. The final program has the lower levels expanded inline like macros where ever they occur.

The top-down, or "stepwise refinement" method proposed by Niklaus Wirth [Wirth 1971a; Wirth 1973; Wirth 1974] has some similarities to both the works of Dijkstra and Mills. Wirth motivates his approach as a method for teaching programming strategy. Each refinement step considers numerous alternatives and then makes explicit the chosen decision. Thus, similarly to Dijkstra, Wirth centers attention on the decision making process and relates each level of the design to a single strategy. (Wirth does mention a tree of "possible solutions" but he sees the programmer's job as the selection of a single path from the root to a leaf of the tree).

Wirth represents each level of the design with a programming language-like text but to which the programmer may add special features when appropriate. He states the following philosophy [Wirth 1971a, p. 227]:
"During the process of stepwise refinement, a notation which is natural to the problem in hand should be used as long as possible. The direction in which the notation develops during the process of refinement is determined by the language in which the program must ultimately be specified ..."

An example, using meaningful variable names and descriptive names for the abstractions, is [Wirth 1971a, p. 223]:

```plaintext
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
    if safe then
        begin setqueen; considernextcolumn
    end else regress
until lastcoldone V regressoutoffirstcol
```

Similarly to Dijkstra there are again two kinds of abstractions, "variables" and the instruction abstractions which are named with the long English strings. Thus Wirth seems to also agree with Mills regarding the need for incorporating at least some sort of description of each abstraction into the text of the level. Wirth further breaks down instruction abstractions into two kinds: "instructions" and "predicates." Predicates return values which can be tested in if and until statements. Neither Mills nor Dijkstra see the need for considering this kind of abstraction separately.

Wirth's abstractions are refined into either programs (in the case of instructions) or traditional data declarations such as integer (in the case of variables) at later levels. Wirth picks levels to make decisions clear and "to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent ..." [Wirth 1971a, p. 221]. Wirth equates teaching of programming with learning the possible kinds of decisions or programming strategies. (Example strategies are preselection, backtracking, and auxiliary variables). He also agrees (with Dijkstra)
that sometimes the best decisions for a given level may not be intuited until lower levels have been considered. Unlike Dijkstra and similarly to Mills, Wirth includes several design decisions or strategy selections in a single level. Wirth also shows an example of a refinement concerned only with reexpressing existing instructions in terms of a newly refined data structure [Wirth 1971a, pp. 223-224]. This kind of level corresponds closely to Dijkstra's joint refinement.

Wirth makes extensive use of English to describe the abstractions at each level. The instructions from the above example (e.g., "consider first column") are each described with a sentence a two immediately following the formal programming language-like description shown here. Although the variables used above do not appear in the formal description of the level, they are used in the English description. For instance, the details of "trycolumn" indicate how the variable "safe" is to be set. Thus, the variables are used to tie together or interface the other abstractions (similarly to Mills' choice of interface at a relatively early level).

In summary, Wirth uses a linear sequence of levels, each one described with both a programming language-like formalism and English text. The levels are selected to untangle and make independent the various design decisions which the programmer faces.

It seems clear that all three of the top-down methods just examined are concerned with two different aspects of program design. First, each details exactly what makes up one level of the design. Second, each describes guidelines for how levels are to be picked and interrelated. The "what" question is by far the easier of the two. The variations
observed among the above methods are things like how many abstractions are defined in a certain level and whether or not English descriptions are to be given. The "how" question usually must rely to at least some degree on the programmer's intuition or good luck. It seems fair to say that guidelines such as "postpone commitments" are sometimes difficult to apply. The next section answers the "what" question for data structure design; consideration of "how" is postponed until Sections 4.4 and 4.5.

But first, there is one remaining work on structured programming which deserves mention since it considers specifically the question of data structures. [Aiello 1974] investigates how well the programming languages PL/I, PASCAL, ELL, and SIMULA (each of which is considered here in Appendix A) can support the semantic data structure model of [Liskov and Zilles 1974] (see Section 5.1.1). This investigation is predicated on the "axiomatic" data structure model (as discussed in Section 5.1.1) and concludes that none of the languages examined are suitable for this method of programming with data structures. This conclusion is based upon both technical and conceptual reasons which are not relevant to the current discussion.

However, Aiello does make some observations on similar topics to those which have been discussed in this section. First he also notes the distinction between "what" and "how;" he states [Aiello 1974, p. 15]:

"Two types of problems which are encountered in building programs structurally may be identified. First, it is evident that decisions concerning the next refinement must be resolved but it is unclear as to how that particular choice is made [i.e., how]. ... The second problem is how to represent the structured program as it develops. This involves representation of both structured control and structured data [i.e., what]."
Aiello assumes the Liskov and Zilles formalism as the answer to "what." He hypothesizes that "how" may be answered at least two ways for data structure design. These two ways are: as soon as possible, or not until it can no longer be avoided [Aiello 1974, p. 18]. The remainder of the paper considers only the "what" question.

4.3 Top-Down Design for Data Structures

Before describing the new top-down method for data structures, the three top-down programming methods surveyed above will be returned to in order to see what each says about data structures. Dijkstra introduces "type" declarations with meaningful names for abstract data structures. He then chooses the true data representation when necessary, i.e., in parallel with refinements of the "instructions." Mills views segments as data transformers; thus, the effect a segment has on a data structure is part of its specification. However, data structures are not actually included in the descriptions of each level. Wirth introduces abstract "variables" with meaningful names and then leaves them abstract as long as possible. He postpones picking the actual data structure as long as possible.

At a recent conference, Dijkstra further pointed out the lack of a top-down mechanism for data structures. He spoke of the need for an "intellectual zoom lens" which would enable a programmer to look at a data structure and see only the necessary amount of detail. The programmer should be able to click down the lens, showing more and more detail until the individual bits of the data are seen [Dijkstra 1974].
has already been presented [Honig 1974]. This paper points out that viewing data structures in a top-down manner is consistent with the basic themes of modern data base management systems (particularly data independence as discussed briefly in Section 2).

Section 4.3.1 develops a top-down design method for data structures. This method provides Dijkstra's intellectual zoom lens for data structures in a way which is complementary to current data base management systems.

4.3.1 A First Example of Top-Down Data Structure Design

With this background, we will now embark on an example top-down data structure design. The new method used is sufficiently general and quite different from any of those mentioned above. The data structure considered is the decision table from Section 1.3. Figure 4-2 and 4-3 are duplicated here from Section 1.3 for convenient reference. Figure 4-2 shows the logical version of the decision table while Figure 4-3 shows it in a form which reflects its implementation using the "coded condition mask" algorithm. The following top-down design describes the decision table in a way motivated by this particular application.

Figure 4-4 shows the beginning of the top-down design. Concentrating for the moment on this first level, level 0, the decision table is described as an ordered collection of rows. Each row is represented by the abstraction DTROW. (The notation "1A" appearing under DTROW means the abstraction is described as the first definition of level 1; the notation is added later after level 1 is finished). The decision to organize the decision table by rows at the first level was made because this most closely models its logical structure as shown in Figure 4-2. The
<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 REGISTER, RELOCATABLE ADDRESS, NO INDEX REGISTER, NO C OPTION</td>
<td>GENERATE LOAD FORM 1</td>
</tr>
<tr>
<td>1 REGISTER, ABSOLUTE ADDRESS, INDEX REGISTER</td>
<td>GENERATE LOAD FORM 2</td>
</tr>
<tr>
<td>RELOCATABLE ADDRESS, INDEX REGISTER</td>
<td>GENERATE LOAD FORM 3</td>
</tr>
<tr>
<td>2 REGISTERS, C OPTION</td>
<td>ERROR TYPE 703</td>
</tr>
<tr>
<td>2 REGISTERS</td>
<td>GENERATE LOAD FORM 4</td>
</tr>
</tbody>
</table>

**FIG. 4-2. LOGICAL DECISION TABLE (COPY OF FIGURE 1-5).**
## Fig. 4-3. Coded Condition Mask Decision Table (Copy of Figure 1-7).

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>GENERATE LOAD FORM 1</th>
<th>GENERATE LOAD FORM 2</th>
<th>GENERATE LOAD FORM 3</th>
<th>ERROR TYPE 703</th>
<th>GENERATE LOAD FORM 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 REGISTER</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2 REGISTERS</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Y</td>
</tr>
<tr>
<td>RELOCATABLE ADDRESS</td>
<td>*</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ABSOLUTE ADDRESS</td>
<td>Y</td>
<td>*</td>
<td>Y</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>INDEX REGISTER</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>NO INDEX REGISTER</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C OPTION</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Y</td>
<td>*</td>
</tr>
<tr>
<td>NO C OPTION</td>
<td>Y</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Y  YES
* DON'T CARE
FIG. 4-4. TOP-DOWN DECISION TABLE DESIGN.
user thinks of the row as the primary entity of the decision table: one row is used to select one possible assembler instruction. Alternative groupings of the information are possible; for instance, the decision table could be grouped by columns with a special column for the action.

As level 0 shows, the components of one level of a design are:

1. Name(s) for the data structure(s) described (here there is one data definition - DECISION TABLE),

2. Abstraction(s) for the components of the data structure described (DTROW),

3. Structuring information in terms of the data structure model (here the aggregate model is used), and

4. Optional commentary in English using is.

Level 0 defines a named data definition in terms of an abstract data structure using the model developed in Section 3. The commentary describes what is assumed about the abstract data structure but does not include how it is to be defined. Comments may also be used for any of the other purposes introduced in Section 3 (e.g., to show the derivation of a name).

This discussion has shown a preliminary answer to the "what" question for data structure top-down design. The use of the various kinds of information in each level can be further understood by analogy with top-down program design. Each level of a program design defines one or more named algorithms or instructions which have been introduced
as abstractions by earlier levels. Likewise, in data structure design each level provides data definitions for one or more abstract data structures which have been named and used earlier. An instruction is defined using a programming language or programming language-like formalism and assuming new abstractions where necessary. For data structures, the data structure model corresponds to the programming language. The model is used to describe a data structure in terms of new abstractions. Finally, both program and data structure design may use natural language commentary and meaningful names to make clear exactly what is being assumed for a new abstraction.

Both processes then introduce new levels to define one or more of the existing abstractions. For program design, this continues until all instructions have been described in terms of the programming language to be used. For data structures the design may similarly be continued until all abstractions are stated in terms of the features provided by the programming language or data base management system which will be used. Alternatively, it may be desirable to design a data structure without assuming any particular programming language or data base management system as the goal. In this case, the basic items of Appendix A serve as convenient primitive items and the design may be terminated when the data structure is expressed entirely in basic items. (Further comments on this difference are postponed to Section 4.5).

Leaving this aside and returning to the top-down decision table design, Figure 4-5 shows the next two levels of the data structure. The decision to be made at the second level is how to represent the abstraction DTROW. Again drawing motivation from Figure 4-2, each row contains any number of conditions and a single action. Thus, a
FIG. 4-5. TOP-DOWN DECISION TABLE DESIGN CONTINUED.
straightforward description of DTROW is the association shown in level 1. This association connects any N CONDITIONS to a single ACTION — two new abstractions for the end data types of the association.

DTROW is given the "code name" or "reference code" 1A since it is the first definition given in level 1. The same reference code would now be added to level 0 (Figure 4-4) to show where DTROW's definition can be found. This cross referencing duplicates the information provided by the names (i.e., the name introduced as an abstraction and defined at some later level). However, the reference code will be much easier to use in large top-down designs. Instead of searching everywhere for a matching name, the reader need only go directly to the proper level and then to the correct ordinal definition. This is particularly helpful when levels use abstractions which have been introduced earlier (as illustrated below in level 4). No existing top-down design methods have included this convenience.

Another special feature of the top-down design method for data structures is the "restatement" shown at the bottom of level 1. The notation "0:1" can be read: "the 0th level, in view of the definitions given at the 1st level, becomes." The purpose of the restatement is to allow the user of a top-down design "to see the forest for the trees." It is helpful every so often to look and see where the design has arrived at; it helps to keep the proper perspective. The restatement shown in level 1 simply states in English the results of the design to date. In other cases a simple picture may be used. Both serve to keep the design (as Dijkstra would say) intellectually manageable, both while it is being carried on and afterwards when it is read.
Mills independently introduced a similar concept and also called it a restatement (see Section 4.2). Mills uses the restatement to show the result of inserting all the abstractions back into the first level. It allows the design to show what the program looks like after processing by the macro substitution implementation technique Mills favors. The data structure restatement used here is very similar except that it is not tied to any specific implementation ideas. The benefits of the data structure restatement are conceptual; they also aid the use of the design as documentation (to be discussed later in Section 4.3.2).

Returning once again to the top-down decision table design, Figure 4-5 also shows level 2. At this level two abstract data structures are described; thus, there are two separate pictorial data definitions shown, headed by the cross reference codes "2A" and "2B." The decision made at this level is to show further details of the decision table's components. The abstractions CONDITION and ACTION are both defined using basic items. CONDITION is defined as a tabular basic item; its definition simply lists all the possible values (the list is abbreviated in Figure 4-5). ACTION is defined as an equivalence of two alternatives, both of which are basic items. Thus an ACTION may be either a particular form of some machine instruction or an integer error number.

At this point the design has reached all basic items. But even though there are no abstractions awaiting refinement, the design can still be continued. Indeed, the design must be continued if it is to describe the data structure in a form which is useful for the coded condition mask technique. The design as of level 2 faithfully represents the decision table as shown in Figure 4-2. Since this form of the decision
table does appear in the assembler's source, the design to this point offers a sufficient level of detail for someone concerned with changing the way the assembler selects machine instructions. However, the further details necessary for an understanding of the assembler's implementation of the coded condition mask algorithm cannot be expressed from the current stage of the design. Instead, a "redefinition" is necessary.

Level 3 of the design, in Figure 4-6, redefines the DECISION TABLE of level 0 as shown. This level represents a different grouping decision than was made at level 0. The same information is present, but at level 3, it is organized differently. The abstractions shown are motivated by the decision table as shown in Figure 4-3. Here it seems reasonable to break down the decision table into three abstractions. CONDITIONS and ACTIONS represent the left-hand and top headings of the decision table of Figure 4-3. USAGE will record the contents of the decision table's rows and columns.

In Figure 4-6, the is comments have been moved outside the actual box of the data definition. This facilitates more complete descriptions of the abstractions; in fact, in a complex data base design as in Appendix C, an entire paragraph may be written to portray an abstraction. The astute reader will also note a slight change in the form of the definition for the hierarchy structure. Previously, the lower part of the box contained two sections: one for the names by which the components were identified and one for the actual data structures used. In doing a top-down design, the data structures used in hierarchy (and other structures) will often be used in exactly one place. In this case, it is practical to use the same name for both the identifier and the type. This convention is akin to Dijkstra's "very wise or very foolish" abbreviation of
**HOMOGENEOUS**: NO  
**BASIC ITEMS**: NO  
**ORDERED**: NO  
**NUMBER**: FIXED, 3  
**IDENTIFICATION**: NAME

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>ACTIONS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CONDITIONS** is conditions appearing in a decision table.  
**ACTIONS** is actions appearing in a decision table.  
**USAGE** is which actions require which conditions.

---

**SEQUENCE**

**HOMOGENEOUS**: YES  
**BASIC ITEMS**: YES  
**ORDERED**: YES  
**NUMBER**: UNBOUNDED  
**IDENTIFICATION**: NONE

**CONDITION**

2A

**USING EARLIER LEVEL**

**SEQUENCE**

**HOMOGENEOUS**: YES  
**BASIC ITEMS**: YES  
**ORDERED**: YES  
**NUMBER**: UNBOUNDED  
**IDENTIFICATION**: NONE

**ACTION**

2B

---

**FIG. 4-6. TOP-DOWN DECISION TABLE DESIGN CONTINUED.**
using type names for variables (as discussed in Section 4.2). Of course, if a hierarchy structure were to use the same abstraction twice then two unique identifiers would be required and the unabridged form of the data definition would be employed.

Returning once again to the design, Figure 4-6 also shows an attempt at level 4. It seems reasonable to attempt to refine some of the components from level 3. First CONDITIONS and ACTIONS may logically be defined as ordered collections of individual conditions and actions (as clearly suggested by Figure 4-3). But then a problem arises: it is not clear how to proceed with the definition of USAGE. A Boolean matrix would suit the information but it is not clear how usable the result would be. The coded condition mask algorithm requires that only rows for false conditions actually be used when the decision table is executed. A more direct relationship between a CONDITION and its section of USAGE would be helpful. Thus it seems that the top-down design has gone astray and a clue must be taken from the intended application of the data structure. To get back on the right track not only level 4 but also level 3 must be discarded and remade.

Thus the top-down design has encountered the same kind of "backup" situation which Dijkstra and Wirth recognize. This sort of backup is to be contrasted with the prior notion of redefinition. In the case of a backup an error has been made, a false path based on a bad decision has been followed. To set the design right, the errors will not be kept; one or more levels will be replaced with new ones representing the proper decision. The levels which are replaced are not kept; they are of no conceivable use to anyone.
Redefinition also remakes some previous levels of the design, but in this case the former versions are retained as a permanent, valuable part of the design. A redefinition represents another, alternative view of the data structure showing additional details. It is not used to eradicate an error. Redefinition adds greatly to the top-down method's ability to completely document a data structure (as will be discussed more fully after completion of the current design).

Figure 4-6 does contain one previously mentioned feature which should be noted before the figure is removed from consideration. The abstractions which CONDITIONS and ACTIONS are defined in terms of by level 4 are not really new abstractions at all! Instead they are reuses of abstractions introduced at earlier levels and, in fact, already concretely defined in level 2. (This situation is analogous to allowing common subroutines in top-down program design.) In this case, the cross reference codes perform a valuable service - they make clear that these are not new abstractions. Obviously, these codes may be added to the level as soon as it is drawn since there is no need to wait on further definitions.

Now, Figure 4-7 shows the new levels which completely replace Figure 4-6. The new level 3 is still a redefinition of DECISION TABLE; it is a different grouping of the decision table's information (different from both level 0 and the former level 3). This new division was stimulated by the problems just encountered down at level 4. Level 3 still uses a hierarchy, but this time one with only two elements. This allows information about the conditions and their usage to be kept together for later use.

Level 4 begins to show how the redefinition of the decision table allows a fuller picture of its intended
HIERARCHY

DECISION TABLE

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 2
IDENTIFICATION: NAME

CONDITIONS | ACTIONS
---|---
4A | 4B

CONDITIONS IS INFORMATION ON CONDITIONS AND THEIR USE IN A DECISION TABLE.

ACTIONS IS ACTIONS APPEARING IN A DECISION TABLE.

4: 4A IS DEFINED

SET

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED
IDENTIFICATION: NONE

ACOND IS A SINGLE CONDITION'S USAGE

5A

3: 4 DECISION TABLE IS A SET OF INFORMATION ABOUT EACH CONDITION AND A SEQUENCE OF ACTIONS.

Fig. 4-7. Top-Down Decision Table Design Continued.
use to be part of the design. Level 4 defines CONDITIONS as an unordered set of elements ACOND; thus, CONDITIONS has been broken down into one element for each condition. This choice represents a decision to organize the usage information on a condition by condition basis as is suggested by the coded condition mask algorithm. CONDITIONS is unordered since the algorithm need not process the conditions in any certain order and is certainly not dependent on the ordering of the left-hand heading of Figure 4-3. Level 4 also defines ACTIONS, in this case in the same way as was attempted in the first version of level 4. ACTIONS is shown to be a ordered sequence of the data structures previously defined in level 2. ACTIONS, unlike CONDITIONS, must be ordered since the decision table includes a preference for its left-hand action.

Level 4 concludes with another restatement of the design since the redefinition of level 0. The shape of the data structure at this point is quite different from the restatement given in level 1. The value of the redefinition capability is that it allows several different, equally useful, views of the data structure to be given. The user of the design need read only so far as to find the amount of detail he/she needs for a given task.

Figure 4-8 shows level 5 of the design which continues with the description of the only remaining abstraction ACOND. The purpose of ACOND is to record everything known about a single condition. The two kinds of information are its name and its usage; level 5 makes the decision to separate them at this point. Thus, ACOND has been broken up into two components: one, CONDITION, was defined long ago and the other, USAGE, is defined in level 6. The purpose of USAGE is to record which actions require that a certain condition be true. A reasonable way to represent this
5A IS DEFINED

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 2
IDENTIFICATION: NAME

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td></td>
</tr>
<tr>
<td>6A,7A</td>
<td></td>
</tr>
</tbody>
</table>

CROSS REFERENCE TO DEFINITION AND REDEFINITION

USAGE IS OCCURRENCE OF Y'S IN DECISION TABLE ROW FOR ONE CONDITION

6A IS DEFINED

FIG. 4-8. TOP-DOWN DECISION TABLE DESIGN CONTINUED.
information is with a "Cardinality" 1-N association. Each condition is related to the actions which require it. Thus level 6 has introduced an association between some existing data structures. Although the two end data structures have existed for a long time, it is not until level 6 that the details of their interrelation were made known. It is interesting to note that the USAGE association is exactly the converse of the DTROW association defined in level 1.

The design has once again run out of abstractions in need of definition. However, the coded condition mask algorithm urges that further attention be paid to the USAGE relation. The algorithm relies upon this information being available in a special form: as a mask suitable for bit-wise ANDing. So level 7, in Figure 4-9, introduces a redefinition of USAGE. This definition describes how the relation of level 6 may be implemented as an array of Boolean basic items. This level also notes, as a comment, how the mask is to be initialized. Finally, level 7 provides a restatement of the entire design. It is interesting to note that even after this long design, the restatement can easily be given with just a few phrases. This fact further adds to the appeal of the restatement feature as a conceptual aid.

Some might wish to argue that level 6 was unnecessary, that USAGE as a Boolean array could have replaced the association used at level 6. Certainly there is nothing to be gained by an overly "deep" design; however, level 6 as stated in Figure 4-8 can be supported on the basis of ease of understanding (which is what top-down design is really all about). The level 7 redefinition is certainly much easier to understand when viewed as the "implementation" of the level 6 association. For instance, the association
6A IS REDEFINED

ARRAY

HOMOGENEOUS: YES
BASIC ITEMS: YES, BOOLEAN
ORDERED: YES
NUMBER: UNBOUNDED, EQUAL TO NUMBER OF ACTIONS (4B).
IDENTIFICATION: NUMBER, 1-N

BOOLEAN

USAGE IS INITIALIZED SO THAT USAGE (1) IS TRUE IF THE CONDITION ASSOCIATED WITH USAGE BY 5A IS RELATED BY 6A TO THE ITH ACTION IN 4B.

O:7 DECISION TABLE IS A SEQUENCE OF ACTIONS AND A SET OF CONDITIONS AND USAGE INFORMATION IN THE FORM OF BOOLEAN ARRAYS.

FIG. 4-9. TOP-DOWN DECISION TABLE DESIGN CONTINUED.
version makes it clear that one condition may be used by any number of actions.

The decision table data structure now rests completely defined in terms of basic items from Appendix A and at a sufficient level of detail to satisfy the coded condition mask algorithm. However, there may still be reasons for carrying the design further. Level 8, in Figure 4-10, shows another redefinition of DECISION TABLE into a form which reflects its implementation in storage using a typical assembly language. Here the entire data structure is defined as three separate ordered sequences. The definitions include the usual sort of information which the lowest level functions in the assembler would need to know to actually access the decision table. For instance, a correlation between the orderings of the condition and usage sequences and the use of the end markers to implement "Number" UNBOUNDED are noted. It is also appropriate at this late level to add implementation restrictions such as the LIMITED "Number" for the action sequence. When introduced here these restrictions will not find their way into higher logical levels of the assembler's organization. If the upper bound on the number of actions need be changed, only the routines which use this final level of the design need be modified. Other parts of the assembler, written to the definition of ACTIONS at level 4 need not be modified — to them ACTIONS is still an aggregate of UNBOUNDED "Number."

4.3.2 A Summary of Top-Down Data Structure Design

Now that the first complete example of top down data design has been carried through to its end, this author appreciates more fully Dijkstra's constant harping about the length of prose necessary to discuss "extensively the
FIG. 4-10. TOP-DOWN DECISION TABLE DESIGN CONCLUDED.
kind of considerations leading to it [the design process]" [Dijkstra 1972, p. 39]. Since many important concepts and features of the top-down data structure design method have been interspersed with the discussion of a particular decision table design, this section first summarizes exactly "what" is meant by top-down design of data structures. Then the philosophies behind this approach are recounted and commented on. Consideration of the "how" question for data structure design is postponed until another example design has been carried out.

Various features of the top-down design method have been introduced in the preceding section; they are:

1. Each level uses the data structure model to define one or more abstract data structures in terms of other named data structures. These other data structures may be new abstractions or previously defined data structures.

2. Comments preceded by "is" may be used to describe abstractions when they are introduced.

3. A "level number - letter" code provides cross referencing between uses of a data structure and its definition (in addition to the connection provided by the data structure's name).

4. A level may provide a redefinition of earlier data structures instead of defining something which is currently an abstraction. A redefinition defines a formerly described data structure from an alternative viewpoint which requires different details or conceptual organization.
5. Any level may contain, in addition to normal definitions and redefinitions, a restatement of some section of the design. A restatement expresses in English or a simple picture a view of some earlier level in light of the further refinements described since its definition.

The philosophies motivating this particular methodology for top-down data structure design are:

1. At each level specific data structuring techniques are picked and described with the data structure model; and

2. The resulting design should provide useful documentation to aid the understanding of the data base.

The first of these philosophies is a direct consequence of Dijkstra’s admonition to be abstract but not vague (see quote from [Dijkstra 1972b, p. 4.8] in Section 4.2). Thus, whenever a data structure is to be defined the designer must pick a specific aggregate, association, or file to implement the abstraction. The particular data structuring technique used is specifically defined with the model. The components of the new data structure may be newly introduced abstractions, but the data structure itself is quite precise. For example, a designer may choose to define the abstraction SKILLS as a set or array containing abstract elements named ASKILL; but he/she cannot avoid the decision of exactly what kind of data structure SKILLS will be. What is not allowed is the introduction of an abstract data structuring technique: in the above example picking SKILLS to be a SKILL-TABLE of ASKILLS. Using a technique such as this, a top-down design would not only be too vague but could also go on forever (since no specific data structures need ever be picked).
The second philosophy arises out of the "intellectual manageability" aspect of structured programming. A top-down data structure design should make it easier for users to understand and use the resulting data base. A large step towards this goal can be provided by intelligible documentation. This documentation must reflect not only the finished product of the design but must also recount the important design decisions made along the way. Thus, the top-down description technique described above has been arranged so that each level of the design is retained as a useful part of the data base's documentation.

Thus, the upper levels of a top-down design using the method described above may represent initial, general, high level views of the data structures. These views are useful to help a person learn and understand the overall organization of the data. The later levels of the design provide more specific details which need be considered only when necessary (e.g., when planning a modification of one part of the data base). The redefinition technique enhances the method's ability to represent both the initial, general design levels and the later, more specific details.

4.4 The "How" Question for Top-Down Data Structure Design

This section describes a top-down design of the scheduling data base of [Frank and Sibley 1973]. While this exercise provides a further example of the methods introduced in Section 4.3, its major purpose is to investigate the "how" question for top-down data structure design. Throughout the design attention is drawn to the kinds of decisions made at each level. At the end of the section the various types of decisions will be summarized.
The scheduling data base has been described in Section 1.3 and used for numerous examples in Section 3. Its major characteristics will be reviewed here in order to motivate the top-down design. The data base is used for scheduling in a manufacturing firm. Information to be represented includes:

1. Information about individual employees,

2. Medical or absence information about employees,

3. Prior job history for each person,

4. Education information for each person,

5. Information about individual machines used in the manufacturing process,

6. Scheduling information, i.e., which people will be working on which machines for some future period, and

7. Skill information; i.e., which skills are possessed by which people and required to operate which machines.

A particular application for this data base is also described in the reference cited above. However, the intent, both here and in the reference, is to define a general purpose data base for the information. This is the kind of effort which a "data base administrator" (as discussed in [Stieger 1970]) would carry out: the result should be useful not only to a single application but to most other conceivable applications as well. This approach is in direct contrast to the decision table's top-down
design (Section 4.3.1); the difference between the two approaches will be discussed in Section 4.5. In fact, the particular application of the scheduling data base discussed in the reference does not require all the information listed above. A solution to this application will be sketched during the top-down design conducted here.

[Frank and Sibley 1973] develops the scheduling data base in terms of the CODASYL DBTG proposal [CODASYL 1971]. This results in a bottom-up approach since "records" must be defined before "sets" can be built. The design carried out here is done in a top-down manner, resulting in a rather different organization. The two results will be briefly compared at the end of this section.

In introducing the scheduling data base design [Frank and Sibley 1973, p. 2] notes that the design is a two step process; the two parts are:

"1. Developing a 'user' data structure which models the information to be stored in the data base as the user sees it, without regard to the capabilities of the DBMS to be used.

2. Converting this 'user' data structure into a data structure whose complexity is within the capabilities of the DBMS that is to be used."

The data structure model and top-down design method of this thesis provide a consistent way of expressing 1) and converting it in an orderly, multiple step process to 2).

Now, without further ado, we shall begin the design. The question is: where to begin? A suitable approach is to pick a partition or grouping of the information required. Taking a hint from Dijkstra's "divide and rule" techniques, it seems that there are at least four basically different
kinds of information in the scheduling data base. They are: information about people, information about machines, scheduling information, and skill information. Figure 4-11 shows the data base broken down into four suitable abstractions. The decision that has been made is one of "grouping;" i.e., deciding what to put with what.

A reasonable decision to be made at the next level of the design is to show the components of the abstractions introduced by level 0. Figure 4-12 presents refinements of PEOPLE, SCHEDULES, SKILLS, MACHINES. In each case, the straightforward choice for their components has been made: the information will be represented in terms of a single individual of each class, grouped together in either an ordered or unordered fashion. For example, PEOPLE is a set of PERSONs and SCHEDULES is an ordered collection of individual SCHEDULEs. In Figure 4-12, SKILL and SCHEDULE are described with "is" comments whereas the PERSON and MACHINE abstractions are sufficiently described by just their names. Thus, each abstraction from level 0 now consists of separate, individual, identical parts.

Level 2 begins with four abstractions still needing definition. At this stage, it is useful to pursue the refinement of some parts of the data base while postponing others. PERSON and MACHINE are both somewhat independent of SKILL and SCHEDULE. On the other hand, SKILL and SCHEDULE both need knowledge about PERSON and MACHINE. Thus, it seems that the next step must be the refinement of PERSON and MACHINE.
HOMOGENEOUS NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 4
IDENTIFICATION: NAME

PEOPLE IS INFORMATION ABOUT ALL THE EMPLOYEES.
SCHEDULES IS THE FUTURE ASSIGNMENTS OF EMPLOYEES TO MACHINES.
SKILLS IS WHICH EMPLOYEES HAVE SKILLS NEEDED BY WHICH MACHINES
MACHINES IS INFORMATION ABOUT ALL THE COMPANY'S MACHINES.

FIG. 4-11. SCHEDULING DATA BASE LEVEL 0.
1A IS DEFINED

PEOPLE

SET

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED
IDENTIFICATION: NONE

PERSON

SCHEDULES

3A, 6B

1B IS DEFINED

SEQUENCE

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: YES,
CHRONOLOGICAL
NUMBER: UNBOUNDED
IDENTIFICATION: NONE

SCHEDULE

1C IS DEFINED

SKILLS

SET

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED
IDENTIFICATION: NONE

SKILL

3B, 6A

1D IS DEFINED

MACHINES

SET

HOMOGENEOUS: YES
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED
IDENTIFICATION: NONE

MACHINE

2B

SKILL IS INFORMATION ON 1 SKILL WHO HAS IT AND WHICH MACHINES REQUIRE IT

SCHEDULE IS WHO IS SCHEDULED TO WORK ON WHAT MACHINE WHEN.

FIG. 4-12. SCHEDULING DATA BASE LEVEL 1.
Figure 4-13 shows the level 2 design; the components of both PERSON and MACHINE are enumerated as hierarchies. It is clear that some of these components could be given in more detail at this point. For instance, PERSONAL DATA could have been expanded into AGE, SEX, BIRTH DATE, etc. However, there seems to be no good reason to introduce more detail than necessary. Instead the remaining specific information has been grouped into the classifications suggested by the original problem statement in [Frank and Sibley 1973], with two important exceptions: ID NUM and MACH NUM have been separated from PERSONAL DATA and MACHINE DATA. The reason for distinguishing them is their key role in the identification of individual PERSONs and MACHINEs. This information is the chief detail needed for the further refinement of SKILL and SCHEDULE.

The choice of ID NUM and MACH NUM at this level also allows the introduction of the PERSON FILE and MACHINE FILE as shown in Figure 4-13. These files provide additional structure which was not apparent at earlier levels. Neither file indicates any sequential ordering since it is not clear (at least at this level) that any ordering is required. The stated application program from [Frank and Sibley 1973] needs the MACHINE FILE. Since a general data base design is being strived for, the similar PERSON FILE has been added.

Level 2 has made the assumption that ID NUM and MACH NUM will be basic items when they are further defined. Both could have been used as "Identification" names for integer basic item hierarchy components at this level. Instead, the choice made is to reserve the right to explicitly define them later. This way the particular kind of basic item need not be picked now; possible choices include integer,
ID NUM is a unique identifier for an employee.
MACH NUM is a unique identifier for a machine.

FIG. 4-13. SCHEDULING DATA BASE LEVEL 2.
string, picture, and tabular basic items. As shown by the cross reference code in Figure 4-13, this choice will not be made until level 8. In Figure 4-13, not all the abstractions have cross reference codes because this particular design is not carried to completion here; instead, some later levels are only partially described for brevity.

Now, the design may consider how SCHEDULE and SKILL fit into the existing design. Level 3, presented in Figure 4-14, depends on the decisions made at level 2. The components shown in Figure 4-14 meet the needs placed upon the SCHEDULE abstraction at level 1. One instance of SCHEDULE describes the employee and machine which will be working for a certain time period. The SCHEDULE FILE, also introduced in level 3, shows a sequential ordering of the schedule information.

The refinement of the SKILL abstraction, also part of level 3, decides how to represent the information about a single skill. A skill is defined by a SKILL CODE, the group of people who possess the skill will somehow be described by the abstraction PEOPLE NOS, and the machines which require the skill will be described by the abstraction MACHINE NOS. SKILL DATA includes any other descriptive information about a single skill. Thus, SKILL is broken down into four components.

Level 4 completes the current train of thought by defining PEOPLE NOS and MACHINE NOS as shown in Figure 4-15. In both cases, the rather simple structure of the unordered set is appropriate since there is no reason to order either the people having a skill or the machines needing it. So PEOPLE NOS is a collection of ID NUMs of people and MACHINE
### 3A is Defined

**Hierarchy**

<table>
<thead>
<tr>
<th></th>
<th>Schedule File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous: No</td>
<td>Selection: None</td>
</tr>
<tr>
<td>Basic Items: Yes</td>
<td>Unique: —</td>
</tr>
<tr>
<td>Ordered: No</td>
<td>Sequential: YES,</td>
</tr>
<tr>
<td>Number: Fixed, 4</td>
<td>Increasing</td>
</tr>
<tr>
<td>Identification: Name</td>
<td>Start Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>8A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Time</td>
<td></td>
</tr>
<tr>
<td>Stop Time</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>Machine NUM</td>
<td></td>
</tr>
<tr>
<td>ID NUM</td>
<td></td>
</tr>
</tbody>
</table>

### 3B is Defined

**Hierarchy**

<table>
<thead>
<tr>
<th></th>
<th>SKILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous: No</td>
<td></td>
</tr>
<tr>
<td>Basic Items: No</td>
<td></td>
</tr>
<tr>
<td>Ordered: No</td>
<td></td>
</tr>
<tr>
<td>Number: Fixed, 4</td>
<td></td>
</tr>
<tr>
<td>Identification: Name</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skill</th>
<th>8B</th>
<th>4A</th>
<th>4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skill CODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has SKILL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needs SKILL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>People NOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine NOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skill DATA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Skill Code** is a unique identifier for a skill.

**People NOS** is group of people who have this skill.

**Machine NOS** is group of machines which need this skill.

**Skill Data** is any other descriptive information about this skill.

**Fig. 4-14. Scheduling Data Base LEVEL 3.**
0 4 SCHEDULING DATA BASE IS FIVE FILES WHICH PROVIDE ACCESS TO PEOPLE'S SKILLS, MACHINE'S NEEDED SKILLS, THE CURRENT SCHEDULES, AND PEOPLE AND MACHINE INFORMATION.

FIG. 4-15. SCHEDULING DATA BASE LEVEL 4.
NOS is similarly a set of MACH NUMs. These components identify a particular instance of either PERSON or MACHINE.

Following the practice of introducing files whenever it seems useful, level 4 also provides two files. Both are "Selection" BASIC ITEM KEY files. PEOPLE'S SKILLS, for example, provides a correlation between an ID NUM and every PEOPLE NOS instance containing it. Since a person may have more than one skill and thus be in more than one PEOPLE NOS instance, "Unique" is NO. For example, a person with two skills will have his/her ID NUM in two different instances of PEOPLE NOS; the PEOPLE'S SKILLS file will select both these instances when that ID NUM is given as a key.

At this point, a solution to the particular application which has been mentioned can be sketched in terms of the design through level 4. The program is to do the following [Frank and Sibley 1973, p. 39]:

"For a specified period of time (in the future), find a person who is capable of running a particular machine, but who is not presently scheduled for that time. Schedule that person to work on the machine."

A top-down program design (in English) for this application follows. Cross references to later levels are provided in parenthesis following key phrases.

0: For each SKILL CODE needed by the given MACH (1A), see if a person with that SKILL CODE is available (1B).
1A: Correlate MACH NUM with MACHINE NOS instances using MACHINE'S NEEDS file. Each instance is a NEEDS SKILL component of SKILL; the SKILL CODEs needed are the corresponding parts of SKILL.

1B: For each person with the SKILL CODE (2A), see if the person is scheduled elsewhere for the specified time period (2B).

2A: The SKILL hierarchy for the particular SKILL CODE provides, in its HAS SKILL component, an instance of PEOPLE NOS for the desired people.

2B: Check far enough through the SCHEDULE FILE to see if the person's ID NUM appears in any instance of SCHEDULE with a conflicting time.

Although this "program" is still at a rather abstract level, it shows how the data structure design can be used. This program structure is also considerably different from the solution in [Frank and Sibley 1973, pp. 32-42].

There are still several abstractions to be defined. However, level 5, in Figure 4-16, instead provides a redefinition. There are two reasons for making some changes at this point. First, while the structures shown for SCHEDULE and SKILL in level 3 are conceptually pleasing and easy to program with, they are not very efficient. The PEOPLE NOS and MACHINE NOS sets duplicate people and machine identifiers several times and would be difficult to update. Second, the design through level 4 has not made use of the kinds of data structures normally provided by network data base management systems.
3 is redefined


**FIG. 4-16. SCHEDULING DATA BASE LEVEL 5.**
So level 5 redefines some of the existing data structures in order to represent the information in a different way. As Figure 4-16 shows, four associations are defined and two new abstractions are introduced. Similar HAS SKILL and NEEDS SKILL associations have been used in examples throughout Section 3; both must be "Cardinality" N-M since, for instance, any number of people may possess the same skill and a person may have any number of skills. However, since an individual schedule only concerns one person and one machine, PEOPLE'S SCHEDULES and MACHINE'S SCHEDULES are both "Cardinality" 1-N. (A person may be scheduled to work on several machines at different times.)

This redefinition accomplished a rather drastic change. The data base is now described in a way such that a program would require considerable navigation to solve the application discussed above.

Level 6, in Figure 4-17, dispenses with the two abstractions just introduced by defining them in terms of their components. Both SKILL and SCHEDULE as defined at level 6 contain a subset of the components they originally had at level 3. The remaining information has been represented by the relations of level 5.

Level 6 also adds additional structure in the form of two files. The SKILL FILE is similar to the PEOPLE'S SKILLS and MACHINE'S NEEDS files of level 4, except that a SKILL is uniquely selected by a SKILL CODE at level 6. The SCHEDULE FILE of level 6 redefines the one from level 3 in terms of the new data structure for SCHEDULE.

There still remains a whole collection of abstractions to be defined by describing their components. Some of these
SCHEDULING DATA BASE IS A NETWORK CONNECTING PEOPLE AND MACHINES THROUGH SKILL INFORMATION AND CURRENT SCHEDULES.
abstractions have been awaiting refinement since level 2. A portion of level 7 of the design is shown in Figure 4-18; other abstractions such as SKILL DATA and MEDICAL would be defined similarly. Next level 8, in Figure 4-19, cleans up the remaining basic items which must be made specific (including some which were just introduced by level 7). Level 8 is also not shown in its entirety; several other abstractions would also be defined as various basic items from Appendix A.

At this point the design is reasonably complete; the scheduling data base has been designed down to a level where it could reasonably be implemented in a data base management system. Additionally, the levels of the design provide clear documentation which is useful at several levels of detail.

Of course, further refinement of the data base may be appropriate in some cases. For instance, if the data base management system to be used does not provide "Cardinality" N-M associations of any sort, another redefinition could be useful. This redefinition could replace the HAS SKILL and NEEDS SKILL relations from level 5 with multiple "Cardinality" 1-N associations. (The method for doing this is discussed in Section 3.3.2).

[Frank and Sibley 1973] present a design for this data base strictly in terms of "Cardinality" 1-N associations (CODASYL DBTG sets). This design differs considerably from the top-down design which has just been carried out; the [Frank and Sibley 1973] design is shown in Figure 4-20 reexpressed by this author as a Data Structure Diagram (see Section 3.3.2 for a model of Data Structure Diagrams and
7: (PARTIAL)

7A IS DEFINED

HIERARCHY

EDUCATION

HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: NO
NUMBER: FIXED, 4
IDENTIFICATION: NAME

DEGREE

START DATE SCHOOL
FINISH DATE STRING

PERSONAL DATA

HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: NO
NUMBER: FIXED, 4
IDENTIFICATION: NAME

NAME STRING
BIRTH DATE AGE SALARY
DATE INTEGER REAL

FIG. 4-18. SCHEDULING DATA BASE LEVEL 7.
8:  (PARTIAL)

8A IS DEFINED

PICTURE
ID NUM
999 X 99 X 9999

8B IS DEFINED

TABULAR
SKILL CODE
1000 ... 1999

8C IS DEFINED

TABULAR
DEGREE
BS, BA, MS
MA, PHD

FIG. 4-19. SCHEDULING DATA BASE LEVEL 8.
FIG. 4-20. ALTERNATIVE SCHEDULING DATA BASE.
Section 5.1.4 for their general description). The details of each of the "record classes" (i.e., names in rectangles) have not been provided in the figure; each consists of the obvious components as used in various places in the top-down design.

The data structure shown in Figure 4-20 is hard to motivate when presented as a fait accompli, but a few comparisons between it and the top-down design are worthwhile. First, SKILL LINK may occur more than once for a given SKILL CODE. This duplication is necessary because the NEEDS SKILL and HAS SKILL sets are both "Cardinality" 1-N and because of restrictions on set instances imposed by the CODASYL DBTG proposal (see [Frank and Sibley 1973, pp. 8-11] for a full discussion). The ramifications of this organization are unclear; at best access becomes more difficult. Second, the PERSON and MACHINE record classes include, respectively, the education and schedule information. The choice between including this information and making it a separate set, as was done for JOB and MEDICAL, is arbitrary. Third, the data structure as shown in Figure 4-20 implies the existence of numerous file structures, as discussed in Section 3.4.2. Some of these files are reasonable and useful; others are forced upon the user and designer because they are implied by the CODASYL DBTG set structure.

This completes the discussion of the scheduling database. It remains to provide the promised summary of the types of decisions made at each level of the top-down design; i.e., to answer the "how" question.
Four different kinds of decisions are evident in the two top-down designs which have been discussed in Section 4; they are:

1. Components decisions,
2. Grouping decisions,
3. Additional structure decisions, and
4. Redefinition decisions.

Each of these will be described briefly and then their use in the two designs will be tabulated.

A components decision provides refinement in the classical sense: an abstraction is broken up into one or more components which are grouped together according to some specific data structuring technique. The data structuring technique is described with the data structure model of Section 3 and the components may be either new abstractions or existing data structures from some earlier level of the design. This sort of refinement is provided for data structures by some programming languages discussed in Section 5.1. The other three kinds of decisions have not been explicitly provided by any current system.

A grouping decision decides what kinds of information to group or collect together. A design starts with some specific information needs; deciding how to structure this information includes picking what to put with what. A grouping decision usually takes the form of an abstraction to represent some class of information.
An additional structure decision introduces further data organization among the existing data definitions. It refines the data base by specifying further details of its structure without introducing new abstractions. The structure added may represent either conceptual organization or logical access paths depending on the level and purpose of the design.

A redefinition decision completely changes the structure of some previous level in order to show more details of the design. The reason for a redefinition is usually to progress from pure documentation or conceptual structure toward an implementation of the data base in terms of some programming language or data base management system.

In the top-down designs carried out here, more than one kind of decision has often been made at each level. This is reasonable since different kinds of decisions are often complimentary. For example, after a components decision has been made additional structure may be obvious.

The kinds of decisions which were made at each level in the scheduling data base design are:

0: grouping
1: components
2: components, grouping, additional structure
3: components, additional structure
4: components, additional structure
5: redefinition
6: components, additional structure
7: components
8: components
The motivation for each decision has been discussed as the design progressed. Level 3 provides some interesting decisions to reconsider here. First the decision was made to refine the PERSON and MACHINE abstractions in terms of their components. In order to do this, a grouping decision had to be made; as a result ID NUM and MACH NUM were singled out at this level while other information which was not yet relevant to the design was grouped together as PERSONAL DATA and MACHINE DATA. Finally, once this grouping had been done the additional structure of the two files could reasonably be introduced.

The same classes of decisions can be recognized in the top-down decision table design in Section 4.3.1. The kinds of decisions at each level are:

0: grouping  
1: components  
2: components  
3: redefinition, grouping  
4: components  
5: components  
6: additional structure  
7: redefinition, components  
8: redefinition

This completes the discussion of the "how" question. The types of decisions identified here must be based upon various real world constraints during a top-down design; thus, simply identifying the decisions types is not a complete answer to "how." Top-down data structure design has not been reduced to a "cookbook" method; indeed, a considerable amount of good judgment is still required of
the would-be designer. The next part of Section 4 investigates some guidelines which may affect the top-down decision making process.

4.5 Other Considerations in Top-Down Data Structure Design

There has been one subtle difference in the decision making processes used in the two top-down designs of Sections 4.3.1 and 4.4. In the decision table design the intended use of the data structure, the coded condition mask algorithm, motivated many of the design decisions. The scheduling data base, on the other hand, was designed with generality and symmetry in mind, ignoring any particular application. This question of generality and a related question about the appropriate goal for the final design level are investigated in this section.

It is reasonable to believe that a top-down data structure design will sometimes be done to create a generalized data structure for multiple uses. It is equally likely that some data structures will be designed for one specific program application. These two different approaches (as well as any approach somewhere between the two extremes) have profound effects on the design and the data structure which results. For example, the data base administrator implementing a corporate data base must design a fairly general structure which is useful for varied applications. The data structure for the decision table, on the other hand, was designed to complement a particular program. The software development data base designed in Section 6.2 assumes a middle ground: it is specialized for a certain kind of programming but not for any one particular application.
Obviously, no one approach to this question of generality is proper in all instances. The designer of a data structure should however be aware of these different approaches and pick the one best suited to the design. Thus in some cases the data structures will be designed first before consideration of any particular program while in other situations one application may be the primary motivation for the design decisions.

Another related question of approach might be stated: How much does where the design is heading affect its top and intermediate levels? In other words, how much should the programming language or data base management system with which the design will be implemented be considered? Any particular system will provide certain data structures and not include others. Of two data organizations which some system does provide, one may be more efficient than the other. Such considerations urge the designer to be aware of the ways the design will be expressed.

However, this question of approach also has a full range of possible answers between two extremes. At one extreme the designer can always be thinking in terms of the desired end and aim to express the information needs as quickly as possible in terms of the data structures provided by some system. This method may result in a design which is not portable to another system and may cause a more elegant design to be missed because it requires an unusual data organization. As an alternative approach, the designer may let his/her imagination run wild and use whatever data structures seem appropriate for the job at hand. This approach also has dangers since the resulting
bottom level data structures may be difficult to implement with reasonable efficiency in a particular system. These worries can be somewhat alleviated if the designer can be assured that the system in use is, in some sense, complete. What must be known is that the normal collection of primitive data structuring techniques will be available in the system.

The top-down designs carried on here have assumed as a base the basic items of Appendix A. This choice corresponds to an assumption that all these structures are either available or easily implemented with the system in use. If this is so, the designer need not conform to any system from the start but can still be assured that the end product of the design will be useable. However, in any case, the question of how much attention to pay to the programming language or data base management system in use is an important guideline for the data structure designer to consider.

One case when this second question of approach is particularly relevant is when the design is not strictly top-down. This may occur when some existing data structures must be interfaced with the new design. Such bottom-up like constraints force the designer to be somewhat aware of a particular goal throughout the design.

4.6 Summary

Section 4 has introduced, used, and discussed a top-down design technique for data structures. This technique describes a data structure with a linear sequence of levels, each level including one or more
data definitions expressed in terms of the model for data structures from Section 3. The components of these data definitions may be either new abstractions or data structures from earlier levels. Abstractions are given meaningful names and described in English. Each level represents one or more decisions about how to show more details of the data structure design. Four different kinds of decisions seem to be identifiable. One of these, the redefinition decision, is unique among existing approaches to both program and data description. The top-down data structure design methodology, particularly as enhanced by the redefinition feature, provides convenient, easy to understand documentation for the resulting data structures.
5. RELATION TO WORK OF OTHERS

Sections 3 and 4 have introduced the two major contributions of this research: a new style of data structure model and a method for using it to describe data structures in a top-down manner. This section compares and contrasts the model and top-down method to similar, existing work. Section 5.1 surveys current models for data structures and develops an enlightening classification scheme for them. Section 5.2 presents claims for the uniqueness and originality of this work.

5.1 Other Data Structure Models

In order to survey the existing models for data structures, this section first develops a simple classification scheme for them. Then each model is described briefly and a common example is expressed, as well as possible, in each of the models.

The classification divides the related work on data structures into three categories. These categories and their definitions are:

**Semantics:** Define a data structure by describing its access functions; these access functions completely characterize the data structure and provide the user's only interface to it.

**Structure:** Describe the static, unchanging, structural aspects of a data structure independently from any access to it.

**Information:** Assume a formalism as the basis for study of real world information. This formalism may have some aspects of a data structure.
The distinction between semantics and structure has already been suggested by both [Mealy 1974] and [Turski 1972]. Turski expresses his philosophy as follows [Turski 1972, p. 288]:

"It is only fair to assume that the problems of data structures admits at least two fundamentally different treatments. One...stems from the programming languages,... The distinguishing aspect of such treatment is the preoccupation with semantic side of problem. ... Second treatment originates in an attempt to describe the morphology of data assemblies, independent of the semantic interpretation and of the processes which may be performed on the data."

One dictionary definition of morphology is "the study of the form or structure of anything." Thus, two common terms for these two areas are semantics and structure; these terms are used here.

Semantic models describe the actual use made of a data structure. They concentrate on the access of the data or the operations to be performed on it. A possible semantic model of an array might be a function which computes the displacement of a particular array element given its index; such an array and three possible models for it are shown in Figure 5-1. The semantic model shown defines a simple access function which takes the index value as a parameter. As another example, a semantic model of a queue would include the information that new elements are always added to the "back" of an ordered collection and that elements to be processed are removed from the "front." Most semantic models do not care about the actual form or structure of their data; data structures which exhibit identical functional characteristics can be treated interchangeably.

Models of structure, on the other hand, are concerned exclusively with form and structure of data organizations.
SEMANTIC MODEL

\[
\text{SALHIST (INDEX)} = \\
\text{IF INDEX} \geq 1 \text{ AND } \leq 5 \\
\text{THEN BASE} - \text{SALHIST} + \text{INDEX} - 1 \\
\text{ELSE UNDEFINED;}
\]

STRUCTURE MODEL (FORTRAN)

\[
\text{DIMENSION SALHIST (5)} \\
\text{REAL SALHIST}
\]

INFORMATION MODEL (ENTITY SET)

\[
\begin{align*}
\text{SALHIST ENTITY} & \quad \text{REAL/PREVIOUS SALARY/1333.33} \\
\text{SALHIST ENTITY} & \quad \text{REAL/PREVIOUS SALARY/1095.00} \\
\text{SALHIST ENTITY} & \quad \text{INTEGER/PRIOR YEAR/1} \\
\text{SALHIST ENTITY} & \quad \text{INTEGER/PRIOR YEAR/2} \\
\end{align*}
\]

FIG. 5-1. THREE MODELS OF ARRAY.
In a structure model, a queue would be just a sequence of elements, similar perhaps to a linked list, and an array is defined simply by declaring its size (see the structure model drawn from FORTRAN in Figure 5-1).

This division between structure and semantics is related to the current distinction in many data base management systems between data description and data manipulation. (This division is typified by the proposed Data Description Language and COBOL Data Manipulation Language of [CODASYL 1971].) Structure models correspond to data definition; the structure or organization of the information is portrayed without providing any accessing tools. The groundwork for access may be laid out, but specific functional characteristics are not defined. For instance, the data definition of a file of information about people might specify several data items, such as name, sex, social security number, and salary, to be grouped together into a "record" describing one person. Further, the data description might state that each value of social security number is unique throughout the file. However, if the distinction between definition and manipulation is strictly adhered to, only the data manipulation language can specify how a value for social security number can be used to retrieve one person's record.

The third category defined above is "information." Models in this category are not really concerned with data structures; instead, they formalize approaches to modeling the abstract information about which users of data base management systems are interested. However, the formalisms developed may be quite like data structures. This distinction between an abstract concept of information and the data which it comes from has also been made elsewhere. For
instance, the IFIP Guide to Concepts and Terms in Data Processing [Gould 1971] includes the definitions:

A1 data A representation of facts or ideas in a formalized manner capable of being communicated by some process.

A3 information In automatic data processing the meaning that a human expresses or extracts from data by means of the known conventions of representation used.

Work classified in the "information" category is concerned with the relation and conversion between information and data in the sense of these definitions. An information model for part of the same array is shown in Figure 5-1; a queue could be modeled similarly with some explicit representation of the ordering between elements.

Thus, the distinction among data structure models is semantics vs. structure vs. information. Some existing work overlaps more than one of these categories. During the discussion of individual models below, some will be placed in more than one category (and the aspects of the work which place it in a particular category will be noted).

A further distinction can be made between models of structure; it is useful to distinguish between "prototype" and "analysis" models. Figure 5-2 repeats the definitions of the three categories of models discussed above and adds definitions of the two subcategories of structure models.

Prototype models use a single, generalized, possibly abstract model which can be used to imitate all data structures of interest. Such approaches postulate one particular structure, which is usually quite different from practical data structures, and by using it attempt to learn things about real world data structuring
SEMANTICS:

Define a data structure by describing its access functions; these access functions completely characterize the data structure and provide the user's only interface to it.

STRUCTURE:

Describe the static, unchanging, structural aspects of a data structure independently from any access to it.

INFORMATION:

Assume a formalism as the basis for study of real world information. This formalism may have some aspects of a data structure.

PROTOTYPE:

Model consists of one formal or abstract construct which can be used to imitate or investigate real world data structures.

ANALYSIS:

Model consists of a compilation of or framework for all possible variations among a collection of real world data structures.

FIG. 5-2. CLASSIFICATION OF DATA STRUCTURE MODELS.
techniques. These models begin with the prototype and show how actual data structures can be simulated.

On the other hand, analysis models attempt to distill or extract from existing data structures a method of modeling them in their full detail. They are in some sense more pragmatic since they represent the characteristics of data structures which have evolved to meet real world problems. Again, resorting to a dictionary definition, analysis is:

"1. The separating of any material or abstract entity into its constituent elements.
2. This process as a method of studying the nature of something or of determining its essential features and their relations..."

Thus, analysis models of data structures attempt to depict the structural nature of existing data structures in order to allow their better understanding and further investigation.

Figure 5-3 shows an analysis and prototype model for the same array modeled in Figure 5-1. The prototype model is a V-graph from the language VERS (see Section 5.1.2); it represents an array as a collection of nodes and named access paths. The analysis model shown is expressed in terms of the data structure model developed here in Section 3; it lists all the array's characteristics, assuming a FORTRAN-like implementation.

Thus, the previous discussion has defined four basic kinds of data structure models. The semantic models are concerned with the operational or functional characteristics of data structures. There are two sorts of structural models, neither of which consider how data is manipulated. The prototype models propose one formal structure which can be used in an illuminating way to describe actual data
PROTOTYPE MODEL (V-GRAPH)

ANALYSIS MODEL (SEE SECTION 3)

ARRAY

HOMOGENEOUS: YES
BASIC ITEMS: YES
ORDERED: YES
NUMBER: FIXED, 5
IDENTIFICATION: NUMERIC, 1-5
REAL IS PREVIOUS SALARY

FIG. 5-3. TWO STRUCTURE MODELS OF ARRAY.
organizations. On the other hand, the analysis models attempt to look at the existing collection of data structuring techniques and model what is already there. Finally, information models provide a formalism for the information content of data structures. Figure 5-2 gives concise definitions of each type of model.

The classification of data structure models developed above, characterizes the basic approaches used to study data. Such studies are carried on for numerous purposes; some of the purposes are:

1. To provide better understanding of data structures and the way they are used; i.e., to communicate better with humans,

2. To describe data at numerous conceptual levels according to the top-down design techniques of structured programming,

3. To describe data at numerous levels, from logical through physical storage, and thereby to attain some measure of data independence,

4. To show the relationship between data structures and other better understood formalisms such as artificial languages,

5. To solve problems of efficient processing and allocation of data structures, and

6. To prove correctness of programs which use data structures.

When possible, the purposes will be noted below when discussing the related works. The purposes of the data
structure model of this thesis are numbers 1 and 2 above; i.e., a data structure model which facilitates better understanding and is also useful for the top-down design of data structures.

Now the existing data structure models will be surveyed, grouped according to the classification just introduced. Figure 5-4 summarizes the other models which are discussed below, grouped according to the classification.

For each model discussed, the data structure presented in Figure 5-5 will be expressed in terms of the model. As shown in Figure 5-5, the example data structure contains basic items, aggregates, an association, and one file; thus, the figure is expressed in terms of all three sections of the data structure model of Section 3 of this thesis. The association HAS SKILL is "Cardinality" N-M so that only one SKILL aggregate is necessary no matter how many people possess any number of different skills. The SKILL FILE selects the instance of SKILL with a given SKILCODE. This data structure is complex: it is an N-M relationship between two different kinds of structures, one of which is connected to a file. Thus, it will serve well to demonstrate the capabilities of numerous different models. The figures presented for each model discussed below follow the pictorial style used in the referenced papers.

5.1.1 Semantic Models

The axiomatic model developed by Hoare [Hoare 1972a; Hoare 1974] represents a purely semantic model of data structures. A data structure is defined by listing its components (i.e., simpler data structures) and defining a set of procedures which are used by any program which
FIG. 5-4. OTHER WORK ON DATA STRUCTURE MODELS.
FIG. 5-5. COMMON EXAMPLE DATA STRUCTURE.
manipulates the data structure. These procedures are completely arbitrary; they are defined in a programming language and operate on the component data structures. The procedures are the only way other programs can access the data structure; any direct access from programs to the components is forbidden. (Hoare suggests the Simula "class" as an appropriate mechanism for implementing these ideas.) The chief purpose of this model is to allow program correctness proofs (purpose number 6 above) according to the "axiomatic" method developed earlier [Hoare 1969]. Toward this end, some "primitive operation" is associated with each of the procedures which define the data structure and it is said that the procedure "models" the primitive function. Programs are then expressed using these primitive functions. Then, once it has been shown that the procedures faithfully model the primitive functions, the way is open for a normal axiomatic proof of the program.

Hoare's axiomatic data structure model also allows the top-down description of data (purpose 2) since the components of a data structure may also be defined in terms of procedures. [Liskov and Zilles 1974] reports a programming language which (unknowingly to its authors - see [Brinch Hansen 1974]) implements this aspect of Hoare's model. The language of Liskov and Zilles allows abstract data structures to be defined as a "cluster." The cluster specifies the next lower level data structures which compose the abstract structure being defined and a set of "operation definitions" which are used to manipulate it. The lower level data structures and their operations are defined in like manner until everything is reduced to the primitive data types and operations provided by the language. The following quote summarizes the philosophy of these first two efforts on a semantic model:
"An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type." [Liskov and Zilles 1974, p. 51]

Figure 5-6 is a partial axiomatic representation of the common example introduced above. The top of the figure lists the eight functions which might be used for the complete skills example. For instance, the PERSON aggregate requires one function for each of its named components and the SKILLS FILE needs one function to supply a correlation between a SKILCODE and the proper SKILL. The axiomatic model forces everything to be a function; no other access is allowed to the information. In the case of the example used here, it would be very convenient if some other simple data structures were also allowed. The lower part of Figure 5-6 shows the lengths to which one must go to express a simple hierarchy as a collection of functions. The "cluster" definition begins by listing the functions which model the data structure (following "is"). Each of these must then be defined in terms of the components of the "rep" (as in representation) of the original structure. Each of the remaining functions listed at the top of the figure must also be defined using appropriate algorithms. For instance, the SKILL FILE probably requires a hashing algorithm. It seems fair to conclude that, at least from an efficiency standpoint, functions are not a suitable replacement for some simple data structures.

The next semantic model considers only a restricted class of data structures: those normally called trees and linked lists. [Shneiderman 1973; Shneiderman and Scheuermann 1974] propose a "Data Structure Description and Manipulation Language" which allows programmers to declare lists (one- and two-way), trees, rings, queues,
FUNCTIONS NEEDED FOR SKILLS EXAMPLE:

\[
\begin{align*}
\text{NAME}(\text{PERSON}) &= \text{STRING} \\
\text{AGE}(\text{PERSON}) &= \text{INTEGER} \\
\text{SSN}(\text{PERSON}) &= \text{INTEGER} \\
\end{align*}
\]

\} \text{ REPRESENT PERSON}

\[
\begin{align*}
\text{SKILCODE}(\text{SKILL}) &= \text{INTEGER} \\
\text{SKLRATE}(\text{SKILL}) &= \text{INTEGER} \\
\end{align*}
\]

\} \text{ REPRESENT SKILL}

\[
\begin{align*}
\text{HAS-SKILL}(\text{PERSON}) &= \text{SET OF SKILLS} \\
\text{POSS-SKILL}(\text{SKILL}) &= \text{SET OF PERSONS} \\
\end{align*}
\]

\} \text{ REPRESENT HAS-SKILL}

\[
\begin{align*}
\text{SKILL-FILE}(\text{SKILCODE}) &= \text{SKILL} \\
\end{align*}
\]

\} \text{ REPRESENT SKILL FILE}

TYPICAL IMPLEMENTATION:

\[
\begin{align*}
\text{PERSON CLUSTER}(\text{N: STRING, A: INTEGER, S: INTEGER}) \\
&\text{IS NAME, AGE, SSN;}
\end{align*}
\]

\[
\begin{align*}
def \text{REP} = (\text{P1: ARRAY}[1..2]\text{ OF INTEGER}; \\
&\text{P2: STRING});
\end{align*}
\]

\[
\begin{align*}
\text{CREATE} \\
&\text{S: REP;}
\end{align*}
\]

\[
\begin{align*}
&\text{S. P1}[1] := \text{A}; \text{S. P1}[2] := \text{S}; \\
&\text{S. P2} := \text{N};
\end{align*}
\]

\[
\begin{align*}
\text{END}
\end{align*}
\]

\[
\begin{align*}
\text{NAME} \text{ OPERATION (S: REP) RETURNS STRING;}
\end{align*}
\]

\[
\begin{align*}
&\text{RETURN S. P2;}
\end{align*}
\]

\[
\begin{align*}
\text{AGE} \text{ OPERATION (S: REP) RETURNS INTEGER;}
\end{align*}
\]

\[
\begin{align*}
&\text{RETURN S. P1}[1];
\end{align*}
\]

\[
\begin{align*}
\text{SSN} \text{ OPERATION (S: REP) RETURNS INTEGER;}
\end{align*}
\]

\[
\begin{align*}
&\text{RETURN S. P1}[2];
\end{align*}
\]

\[
\begin{align*}
\text{END}
\end{align*}
\]

\[
\begin{align*}
\text{PERSON}
\end{align*}
\]

FIG. 5-6. LISKOV AND ZILLES' LANGUAGE FOR AXIOMATIC MODEL.
stacks, and deques. These data structures can be manipulated using built-in insert, delete, detach, copy, interchange, replace, and search operations. These operations insure that the semantics of a particular data structure will be obeyed. For instance, new elements can be inserted only at the back of a queue; any attempt to do otherwise will cause a run-time error. This semantic model is quite different than the above one due to Hoare; it provides a built-in, predefined definition of the semantics for a particular class of data structures. Thus, the programmer who is content to use only the provided data structures need not spend additional time programming characteristic operations for each data structure; these defining operations are in effect built into the implementation of Shneiderman's operations. Since these built-in operations are part of the system, they can be assumed correct by the programmer; hence, the correctness proof for the program using them is simplified (purpose 6 above).

The Data Structure Description and Manipulation Language also allows the declaration of "multistructures," or data structures whose elements are also nonatomic data structures. For instance, a one-way list of trees of stacks could be defined. This capability supplies a sort of top-down method (purpose 2 above): at one level of a program, a list can be manipulated while at a lower level the list elements could be accessed as trees. In addition, the built-in operations make sure a multistructure will remain well formed. In the case of the one-way list postulated above, the system would prevent the insertion of a queue into the list because the declaration specifies the list elements to be trees.

Figure 5-7 shows an attempt at representing the common example in Shneiderman's model. Due to the nature of this
ENTRY NODE

ONE PERSON

THE PERSON'S SKILLS

LAST-PERSON

PEOPLE

HAS-SKILL

NEXT-PERSON

HAS-SKILL

NEXT-PERSON

HAS-SKILL

DECLARE SKILLS-EXAMPLE

• (HOST LANGUAGE ENTRY MODE DEFINITION)

LEVEL(1) LIST (PEOPLE) ONEWAY (NEXT-PERSON) END (LAST-PERSON)

• (HOST LANGUAGE DEFINITION OF PERSON)

LEVEL(2) LIST (HAS-SKILL) ONEWAY (NEXT-SKILL)

• (HOST LANGUAGE DEFINITION OF SKILL)

FIG. 5-7. SHNEIDERMAN'S STRUCTURED DATA STRUCTURE.
model, the example shown is very list oriented. Unfortunately, the model cannot faithfully represent the "Cardinality" M:N association; the best which can be done is to change the example to a "Cardinality" 1:M association and allow each person to have their own, individual collection of N skills. Thus, a particular skill may appear many times, once for each person who has it. The same modification will be made whenever any of the models cannot handle the more complex form of association. Thus, Figure 5-7 shows a list of people and each person contains a sublist of skills. Note that (1) there is no way whatsoever to represent the file (the search statements provided by this model are only navigational), (2) the model considers only the linkages - there is no provision for defining the ends of the association, and (3) this example does not illustrate any of the access and manipulation operations provided by the model.

Work of Jay Earley on the VERS programming language [Earley 1971] is chiefly of a prototype nature and will be mentioned below. However, one aspect of VERS is similar to Hoare's axiomatic or functional model. VERS allows the definition of V-graphs (see below) and "transformations" on them. A transformation changes a V-graph, perhaps adding or deleting elements, and hence is like a specialized version of Hoare's procedures (which may simply access as well as modify a data structure). These transformations are defined in VERS and, together with a description of the initial configuration of a V-graph, are viewed as describing an entire class of V-graphs. The class consists of all V-graphs which can be obtained from the initial configuration by applying the transformations. Thus, the transformations define the semantics of a class of V-graphs.
5.1.2 Prototype Models

The first prototype model to be considered is the VERS programming language [Earley 1971] whose semantic aspects were mentioned above. The language is based on "V-graphs" which are a formal way of describing the structure of many common data structures; their purpose is to allow a better understanding of what data structures such as arrays and linked lists really represent (number 1 above). A V-graph is essentially a directed graph with named edges. The nodes of the graph which have no out-bound edges represent information to be stored as part of a data structure; these nodes or "atoms" have traditional types such as integer, Boolean, string, etc. The other nodes serve only to connect various edges. The edges or "links" represent the structure or organization of the information; each edge is named and there is at most one link from any node with a given name.

Thus, rather simple prototype models can imitate a wide variety of real world data structures. For instance, a traditional 5-element array could be modeled as a V-graph of one nonatomic node linked with edges labeled 1, 2, ..., 5 to five atoms (see Figure 5-3).

The common example expressed in VERS is shown in Figure 5-8. The nature of this prototype model forces a similar linkage-like implementation of the aggregates, association, and file. The true "Cardinality" N-M association can be represented, in this case, due to the ability to declare any number of links with names of a given type from a single node (e.g., each SKILL node may link to any number of PERSONs along an access path named from the type INTEGER). VERS thus successfully models the entire example data structure, but forces it into a particular representation as a V-graph.
TYPE SKILL-FILE IS [INTEGER] TO SKILL

TYPE SKILL IS

  1 TO INTEGER
  2 TO INTEGER

[INTEGER] TO PERSON

TYPE PERSON IS

  SSN TO INTEGER
  NAME TO STRING
  AGE TO INTEGER

[INTEGER] TO SKILL

FIG. 5-8. EARLEY'S VERS.
The philosophy behind the next prototype model to be considered is similar to the distinction between semantics and structure discussed above. The "data graphs" of Rosenberg [Rosenberg 1971; Rosenberg 1972] are also directed graphs. Their purpose is to study "properties of data structures which depend only on form and not on meaning. ...[A] data graph is obtained from a data structure by masking out the data items which appear at the nodes of the structure" [Rosenberg 1971, pp. 193-194]. Despite this quote, data graphs are not selected from real word data structures; instead, they are defined mathematically as an ordered pair of "data cells" and "atomic link transformations" which in effect form a directed graph. The edges of the graph are not labeled with names (as in VERS) but instead with "relations" such as greater-than or other mathematical functions. Additionally, data graphs must satisfy the "strong connectivity axiom"; i.e., there must be a path between any two nodes. Thus, data graphs represent a rather abstract and restricted class of data structures.

However, these restrictions facilitate the use of data graphs to investigate several practical questions relating to the implementation of data structures. The first question considered by Rosenberg concerns the placing of data graphs into an "address space" (computer memory) using either "relative addressing" (similar to sequential storage allocation) or "relocatability" (resulting in a data structure which is independent of the addresses at which it is stored). In both cases, certain additional restrictions are developed and it is shown that a data graph can be "realized" by relative addressing (or relocatability) if and only if it is "addressable" (or "free-rooted"). Later work has attacked the question of how to realize arrays which
vary in size [Rosenberg 1974]. Thus, the purpose of this work is to investigate certain practical problems with the aid of a formal model of data structures (purpose 5 above).

Unfortunately, some of the restrictions on this model prohibit its application to data structures such as our example. Because of the strong connectivity restriction, for example, it is impossible to model the case of two people, each with a single (different) skill. Data graphs are best suited to symmetrical, uniform, infinite data structures.

The next prototype model is also based on a mathematical formalism which characterizes a somewhat restricted class of data structures as an n-tuple of mathematical objects. However, in this case the purpose is to show a relationship between this specially defined type of data structure and formal languages (number 4 above). Once the relationship is proven, the numerous formal results from artificial language theory can be applied to data structures. [Fleck 1971] defines a formal model for data structures which are similar to linked lists and directed graphs. The data structures defined are "simple list," "list set," and "list structure." List structures are a suitable model for directed graphs with loops and recursion, such as the "Lists" of [Knuth 1969, pp. 312-314]. The main formal result is a proof that context free languages and list structures (as defined) are equivalent. This result allows immediate conclusions such as there is no list structure corresponding to the language $A^nB^nC^n$, $n = 1, 2, ...$ (which is not context free).

Figure 5-9 attempts to model the common example as such a list structure. A directed graph can easily implement the "Cardinality" N-M association, but the file
FIG. 5-9. FLECK'S LIST STRUCTURE.
structure cannot be represented. Note that the list structure imposes an unnecessary ordering on each person's component data elements (i.e., first NAME, then AGE, then SSN). The formalism proposed by Fleck is shown at the bottom of the figure: each simple list node consists of a node name or number, data or reference to another list, and the name of the next node in the current list. Fleck's list structures do a good job of modeling directed graphs; however, they provide just a single data organization.

Earley has forsaken his work on VERS (discussed above) for another prototype model which provides three levels and certain different prototypes at each level. His new model, which is also the basis of a (currently unimplemented) programming language VERS2 [Earley 1973], thus allows a limited form of top-down design. However, the main purpose of VERS2 is to improve data independence (purpose 3) by providing "relational," "access path," and "machine" levels of data structure description. VERS2 contains many interesting features (including control structures specially designed for use with relational data structures); however, the discussion here will consider only its data structures. There are four separate prototype data structures which are provided by VERS2. These data structures are used with full freedom at the highest, most abstract relational level. At lower levels, certain restrictions apply. The four data structures are all mathematically-inspired; they are: tuples, sets, relations, and sequences (all used quite truly to their mathematical definitions). A complex data structure may be represented by nesting, e.g., a relation over tuples. (Similarly to Shneiderman's work discussed above, this mechanism allows some aspects of top-down design.) Unlike the works of Rosenberg and Fleck, the VERS2 prototypes can
probably be used to imitate the full range of real world data structures.

VERS2 can easily represent most aspects of the common example; Figure 5-10 shows the example modeled with the relational level data structures of VERS2. The tuple and sequence structures are straightforward analogs of traditional hierarchies and (unbounded) arrays. SKILL FILE is implemented as a special sort of binary relation which defines a mathematical function from its first domain to its second. HAS SKILL is a more general relation which does not reduce to a function. In VERS2 notation, HAS-SKILL(a person, *) is the N skills the person has; likewise, HAS-SKILL(*, a skill) is the M people having the skill. Unfortunately, this structure implies that each skill appears once for each person having it and each person appears once for each of their skills.

One part of the "Data Independent Accessing Model" [Senko et al. 1973] is a model of abstract information and will be discussed below; however, the other aspects of this work can be viewed as a prototype model with motivation similar to Earley's VERS2. The Data Independent Accessing Model (DIAM) is proposed as the basis of a data base management system which provides data independence using a four level approach to data description. The "entity set model" is the highest level and is the information model which will be discussed later. The common example will be expressed in terms of this level. The "string model," "encoding model," and "physical device level model" provide three levels of prototypes for encoding "entities" and their interrelationships. The string model is similar to Earley's earlier VERS language and to the "access path" level of VERS2; it consists of "atomic strings," "entity strings," and "link strings," all of which
PERSON :: < NAME -> STRING, AGE -> INT, SSN -> INT >

SKILL :: SEQ(INT)

DECL SKILL-FILE -> FUNC(INT, SKILL)

DECL HAS-SKILL -> REL(P -> PERSON, S -> SKILL)

FIG. 5-10. EARLEY'S VERS2.
link together individual pieces of information in particular ways. The encoding level describes the bit or byte level encoding of the string model, and corresponds roughly to the machine level of VERS2. The encoding level expresses everything in terms of a single prototype called a "Basic Encoding Unit" (BEU). BEUs are thought of as residing at "reference addresses" within "named address spaces." At this level, the data structure is viewed as consisting of noncontiguous blocks of sequential addresses. Finally, the physical device level specifies how these BEUs are actually allocated to real computer storage such as disks and drums. For instance, blocks, tracks, cylinders, and overflow records on disks are considered at this level.

5.1.3 Analysis Models

[Hsiao and Harary 1970] present an analysis model based on inverted, index-sequential, and multilist files as they are commonly used in document retrieval applications [Lefkovitz 1969]. The approach is based upon concepts of "attribute," "value," "record," "index," and "address," which are distilled and generalized from practical applications. The result is a "generalized file structure" and its "directory" which are defined as mathematical sets and sequences of the terms mentioned above. By restricting the size and composition of these sets and sequences, each of the real world file structures mentioned above can be modeled. The generalized file structure then provides a concise description of the structural organization of such real world files. The classification of this work as an analysis model is slightly questionable; in a way, the "generalized file structure" is a prototype since it is a single, generalized, abstract construct which can represent various file structures. However, since the generalized file structure faithfully represents all the
typical variations among real world files, it can properly be classified as an analysis model.

Hsiao and Harary's model is, of course, intended only to represent file structures; thus, it cannot completely portray our common example. Figure 5-11 shows a representation of the example which includes as much of the intended structure as possible with a generalized file. Only the "keywords" or the words upon which the file is indexed can be shown; the remaining components of PERSON and SKILL are ignored. The data structure shown combines the information from the SKILL FILE and HAS SKILL association into a single generalized file structure. The directory and file correlate either a name or a skill code with any number of "records." To find all the people with a skill, the directory yields the address of the start of a list through all the desired people. Once again, the "Cardinality" N-M association has had to be sacrificed in favor of allowing a skill to occur once for each person having it and a person to occur once for each of their skills. It should be noted that the generalized file model also allows consideration of numerous different implementation techniques for files.

Smith has developed a model for data items and structures used in data base management systems [Smith 1972; Prywes and Smith 1972]. This model has three levels, the "record," "file," and "storage" levels, and in this way is similar to the works of Senko and Earley (VERS2) discussed above. However, the model produced is of a basically different nature. At each level, the model is simply a detailed specification of each of a set of "characteristics." For instance, the record level consists of 13 characteristics; some of these characteristics are: character codes, length, length
FIG. 5-11. HSIAO AND HARARY'S GENERALIZED FILE.
uniformity, and repetition number. The claim for this model (supported by a "completeness" proof) is that these characteristics allow the representation of all possible record level data structures. This approach should be contrasted with any of the prototype models discussed above which, instead, provide just a single structure and use it to imitate actual data structures.

Figure 5-12 shows the common example expressed in the record and file levels of Smith's "Generalized Data Description Language" [Smith 1972, app. A]. The various characteristics are simply strung out as parameters to each kind of declaration. Most of the 13 record level characteristics mentioned above appear in the "group" and "field" declarations shown in the figure (the other characteristics are optional and not appropriate to our common example). The "field" AGE, for instance, specifies the following characteristics:

- field name: AGE
- character code: EBCDIC
- length type: C, i.e., length is a number of characters
- length: 3
- uniformity: V, i.e., variable length
- data type: N(10,NS,FX), i.e., base ten, no sign, fixed point

Thus, the details of the lower level components of the example can be quite fully expressed. There is no way to model the SKILL FILE; the model does not consider any other than simple files. Such files are modeled in terms of their "links" - a rather implementation oriented concept of a directed access path. In order to link a person with his/her skills, the SKILCODE field must be introduced so that each SKILCODE is physically present in the records of the people possessing it. Then the linkage "criterion" can state when such a path should be provided between two
FILE ('SKILL EXAMPLE', 'HAS SKILL', 'POSS SKILL', ...)  
LINK ('HAS SKILL'; 'PERSON'; 'SKILL'; 'CRIT1'; EMBED, 'HSP'; 1, F)  
LINK ('POSS SKILL'; 'SKILL'; 'PERSON'; 'CRIT2'; EMBED, 'PSP'; 1, F)  
CRITERION ('CRIT1', ('SKILCODE' OF OCC('PERSON' H)) EQ  
   ('INT'(1) OF OCC('SKILL' T)))  
CRITERION ('CRIT2', ('INT'(1) OF OCC('SKILL' H)) EQ  
   ('SKILCODE' OF OCC('PERSON' T)))  

FILE CONSISTS OF TWO KINDS OF DIRECTED LINKS, HAS SKILL AND  
POSS SKILL, BETWEEN RECORDS OF TYPES PERSON AND SKILL. THE  
CRITERION FOR HAS SKILL STATES THAT THE LINK EXISTS IF SKILCODE  
IN THE PERSON RECORD AT THE HEAD OF THE LINK EQUALS INT(1) IN  
THE SKILL RECORD AT ITS TAIL.  

RECORD ('PERSON', 'PERGROUP')  
RECORD ('SKILL', 'SKGROUP')  
GROUP ('PERGROUP', NOORD, ('NAME', M, 1, F), ('AGE', M, 1, F), ('SSN', M, 1, F),  
   ('SKILCODE', O, NOLIM, V; O, ASCEND))  
GROUP ('SKGROUP', SPEC, ('INT', M, 2, F))  

PERSON IS AN UNORDERED COLLECTION IN WHICH IT IS MANDATORY THAT  
NAME, AGE, AND SSN OCCUR EXACTLY ONCE AND IN WHICH ZERO OR  
MORE SKILCODE'S MAY OCCUR IN ASCENDING ORDER.  

FIELD ('NAME', EBCDIC, C, NOLIM, V, C)  
FIELD ('SSN', EBCDIC, C, 9, F, N (10,NS,FX))  
FIELD ('SKILCODE', EBCDIC, C, 4, F, N (10,NS,FX))  
FIELD ('INT', EBCDIC, B, 32, F, N (10,NS,FX))  
FIELD ('AGE', EBCDIC, C, 3, V, N (10,NS,FX))  

AGE IS UP TO 3 EBCDIC CHARACTERS REPRESENTING AN UNSIGNED, FIXED  
POINT, BASE 10 NUMBER.  

FIG. 5-12. SMITH'S GENERALIZED DATA  
DESCRIPTION LANGUAGE.
records. Thus, the "Cardinality" N-M HAS SKILL association is implemented with multiple "Cardinality" 1-1 linkages.

Section 3 of this thesis introduced a new analysis model which covers a different range of data structures than those considered by Smith and Hsiao and Harary. Section 5.2 compares this new model to the other data structure models surveyed here in Section 5.1.

5.1.4 Information Models

The information models arose from data base management. They are attempts to model abstract information and represent it in a form which is computer tractable.

The relational model treats all information as mathematical relations on objects from specific "domains" or sets; the data structure based upon this idea was covered in Section 2.2.2.1. The approach has been formulated by Codd and associates [Codd 1970; Codd 1971a]. The basic claims of this approach are that all common kinds of information (or at least those normally used in data base management systems) can be viewed as relations [Codd 1971] and that the resulting formalism is quite simple and easy to use [Date and Codd 1974]. The interrelation between information modeled in this way is represented by numerous operations on the relations. Such operations may be traditional mathematical ones such as union and intersection or specialized relational operations such as "projection" and "join" [Codd 1972].

The common example in terms of the relational model is shown in Figure 5-13. The relations shown here are expressed in "third normal form" which provides access and update advantages [Codd 1971; Codd 1972a]. The
FIG. 5-13. Codd's Relational Model
requirements of third normal form force the "Cardinality"
SKILL-HAS-SKILL association to be broken up and expressed in
terms of CRN and SKICODE as shown in the figure. There
is no way to explicitly model the SKILLS FILE here; instead,
the user of the relational model has complete freedom of
access. Any domain can be used as a "Selection" BASIC ITEM
KEY file and the underlined domains or "primary keys" also
have "Unique" YES.

The next information model is the Entity Set Model
developed in [Senko et al. 1973]. (The Entity Set Model
and related concepts are discussed here; some other aspects
of DIAM were described above under prototype models.) The
Entity Set Model proposes a basic formalism for the informa-
tion to be stored in a data base management system and is
thus similar in nature to the relational model. In fact,
there seems to be a rather close correspondence between
numerous concepts in both models. This correspondence has
not been noted in the literature; it will be pointed out in
the following discussion.

The Entity Set Model begins by formalizing the basic
primitive quanta of information which are called "entities."
A person or a part is an entity. Entities may also be
terms or concepts which describe other entities, e.g., the
color red. Associated with each entity is one or more
"names" or identifying symbols. A collection of these
names may be grouped together to create an "entity name set"
(corresponding to a "domain" in the relational model) which
describes a collection of things in which the data base
management system is interested. Entity name sets are
labeled with "entity name set names" and are referred to
by "role names." (Role names correspond to the relational
model's "attribute names."). Sets of triples consisting of
an entity name set name, role name, and entity name are
called "entity descriptions." They form a complete description of one entity and show its relation to other entities; entity descriptions correspond to the traditional data base management system's records. Two example entity descriptions are shown in the balloons at the bottom of Figure 5-1. A subset of the entity name set name and role name pairs which uniquely name each entity is called the "identifier" (and is similar to a "primary key" in the relational model). Finally, a set of uniquely named (by identifier) entity descriptions forms an "entity description set" (corresponding to a "relation"). Two different role names in entity description sets may refer to the same entity name set allowing associations between entity descriptions using the same entity names.

Users of DIAM assume the entity set model as a representation of real world information; programs can be written in terms of the names and interrelations defined as described above. A system catalog stores these definitions and also the description of the three lower levels discussed in the section on prototype models. The system will automatically translate the data independent entity set level programs into appropriate terms of the lower levels. Programs can also be written to use the string level model directly.

Figure 5-14 presents the common example as an entity set model. A single entity in this version of the example corresponds to one person and one skill; the "cardinality" N-M HAS SKILL association has been abandoned. The entity set model also provides no special facilities for representing the SKILLS FILE. In fact, the information organization shown in Figure 5-14 bears very little resemblance to the original structure of our example.
FIG. 5-14. SENKO'S ENTITY SET MODEL.
The final information model to be discussed was developed as a notation and documentation aid for database designers using the early database management system IDS [Bachman and Williams 1964]. "Data Structure Diagrams" [Bachman 1969; Eriksen 1974] are a graphic technique for describing owner-member structures between "entity classes." The association model of this thesis was tested against Data Structure Models in Section 3.3.2. An entity class is similar to the Entity Set Model's entity set; possible entity classes are people and parts. Entity classes are represented by blocks; the blocks are connected by arrows which represent "set classes." These arrows point from the "owner" to the "member" of the set class. Each owner represents exactly one instance of an entity class; zero, one, or more instances of the member are allowed. A few extensions are added to allow "sometime membership," "multimember set class," and "alternate owner set class." The owner of one set class may be the member of another (and vice-versa) so that, in conjunction with the extensions, rather large and complex information models can be built.

Data structure diagrams thus can model data organizations built-up out of "Cardinality" 1-N associations. As explained in Section 3.3.2, such combinations can simulate "Cardinality" N-M associations; this approach has been used in Figure 5-15 to model the common example. Data Structure Diagrams do not offer any way of representing either the components of each entity class (e.g., PERSON) or the selection supplied by the SKILLS FILE.

The survey of related work on data structure models and the exercise of expressing a common example in terms of each of them is now complete. Section 5.2 contrasts the model of this thesis with these others.
FIG. 5-15. BACHMAN’S DATA STRUCTURE DIAGRAM.
5.2 Uniqueness of This Work

This section discusses the distinguishing qualities of the work reported by this thesis. The uniqueness of the data structure model is described first, then the special features of the top-down data description method are discussed.

The data structure model presented in Section 3 uses a different approach and covers a different scope of data structures than any of the other model's surveyed in Section 5.1. The data structure model developed here is an analysis model for the data structures commonly used in programming languages and data base management systems. The key words are "analysis" and "commonly."

The data structure model is unique from the other works classified as semantic, prototype, and information models in Section 5.1 because of its different purpose and approach. The data structure model, as an analysis model, provides a framework for understanding the variations among real world data structures. This purpose is quite distinct from the other kinds of models which avoid questions arising from data structures as they actually exist in today's systems.

Second, none of the other analysis models apply to the class of data structures which the model presented here considers. This class includes all the data structures tabulated in Appendix A (and surveyed in Section 2); these data structures are the ones commonly in use by today's programmers and data base designers. The other two analysis models which currently exist are the generalized file of Hsiao and Harary and the data translation model of Smith (both described in Section 5.1.3). Hsiao and Harary's
model considers only the file data structures (including some details of their implementation). Smith's work is aimed toward the translation of data bases from one hardware system to another; thus, it considers rather low level data structures which normally occur in production data base management systems. These data structures are mostly the basic items of Appendix A plus traditional concepts of records and repeating groups. Thus, neither of these other analysis models can represent the full spectrum of data structures which the model of this thesis includes. In fact, Section 5.1 has shown that many of the other models in all four classifications have trouble modeling the example skills and people data structure without modifying it. Thus, very few models in any of the classifications cover the wide range of data structures included in this new data structure model.

The top-down data structure design methodology introduced in Section 4 includes some unique features which make it particularly useful for the design and documentation of large data bases. In addition, the "how" question as discussed in Section 4.4 has never before been considered exclusively for data structures.

The top-down method developed here includes three special features: redefinition, restatement, and cross reference. These features and the ways they aid documentation and design have been discussed in Section 4.3.2.

The redefinition concept is unique; it is a departure from existing methods of (both data structure and program) top-down design. Existing methods allow only a strict, tree-like refinement at each level of the design. The $n + 1$st level simply defines abstractions from the $n$th or earlier levels. The top-down method of this thesis
recognizes the need to allow complete alteration and reexpression of preliminary levels; this need is filled by the redefinition. The fact that such need exists is best demonstrated by the examples of Sections 4.3.1 and 4.4; both of these designs would be nearly impossible to express in an understandable manner without the redefinition feature.

Figure 5-16 illustrates in outline form the role which redefinition and also restatement play in top-down designs. The redefinition shown at level 3 in the figure reexpresses two of the abstractions originally defined at level 1 in terms of a new, single definition. Two separate subtrees of the design are thus brought together and defined in terms of different details from level 4 onward. Such redefinitions are necessary for drastic changes in a data structure design, e.g., conversion between logical and actual data organization. However, redefinition is also helpful just for understanding: to express a data structure first in a manner which shows its purpose and then to re-express it in a form more suitable to the rest of the design.

Figure 5-16 also shows, in level 3, a restatement. The restatement makes it easier to understand where the design has gone from its root at level 0 down through level 3. The restatement, in effect, summarizes one path from the root to one node. As discussed in Section 4, restatements were introduced independently by Mills. Mills, however, sees restatements simply as a replacement for an automatic implementation facility. The research of this thesis recognizes the restatement as a valuable mental aid which helps to keep the design intellectually manageable.
The "level number - letter" cross reference scheme used in the designs of Section 4, while not particularly startling, is also original. The only other attempt at any sort of cross reference not based solely on names is the rather confusing one introduced by Dijkstra in [Dijkstra 1972, pp. 31-38]. The simple cross reference method used here is quite valuable particularly in large designs such as the one shown in Appendix C.

These three features and the intrinsic style of the new top-down design method make it useful not only for design but also for documentation of data structures. Traditional approaches to design and documentation of large data bases quickly result in understanding being lost among too many details. An example of such a traditional design gone bad is [National Health Computing Services Group 1972]. A top-down design can, instead, present an abstract, conceptual level to which details are added as refinements. With this sort of design, each step can be retained as documentation; then, understanding can also proceed from relatively simple, higher levels to the details as necessary.

Some of the other data structure models surveyed in Section 5.1 were noted as providing some top-down capabilities. These approaches, however, model only those levels of the data structure design which are actually to be used by programs at run time. The major advantages of structured programming are conceptual; a top-down design of data should also help a person learn and understand its structure. The top-down approach developed here is also able to represent initial, general levels of the data design and, hence, be a valuable source of program documentation.
This difference between views of top-down design has also been noticed by Denning, with regard to top-down design of programs. In reference to papers containing example "well-structured" programs which he reviewed [Denning 1974a, p. 5] states:

"Somehow, these authors did not grasp the importance of the central idea advanced by Dahl, Dijkstra, Hoare, and others: namely, one is always working with abstractions. It is insufficient to present the end-product and expect the beholder to perceive its structure by inspection, or even deep motivation. Instead, the beholder must also see at least part of the programmer's thought processes, starting from the original (very abstract) version and proceeding to the end-product via a clearly presented sequence of clear transformations and refinements."

The top-down data structure design method introduced here provides a way of recording the data structure designer's thought processes beginning at the most abstract stage of the design.

Finally, this work is unique in its attention to the "how" question for data structure design. Detailed consideration of the "how" question is rare even for top-down program design. The only similar emphasis is Wirth's attention to the teaching of programming strategies (see the description and references in Section 4.2).

In conclusion, the work of this thesis is unique because it provides a new way of understanding and using a wider range of data structures than any existing work.
6. **ARGUMENTS FOR UTILITY**

Section 6 presents three arguments for the usefulness and utility of the research described in the previous sections. These arguments demonstrate the value of both the data structure model (of Section 3) and the top-down data structure design method (of Section 4).

The three arguments are:

1. The data structure model is complete in the sense that it models the common data structures,
2. The top-down method and the model are useful for describing real world data bases, and
3. The model is understandable and easy to use.

Each of the following subsections argues one of these claims.

6.1 **Completeness**

This section will demonstrate the completeness of the data structure model. Completeness is used here in a very pragmatic sense to mean that the model is sufficient to represent common, practical data structures. The data structures compiled in Appendix A are the ones which a wide variety of system designers deemed useful; thus, it is reasonable to test the completeness of the data structure model of this thesis by seeing if it can model this collection of data structures. The full enumeration of each individual data structure and its variations appears in Appendix B. This section introduces the method used in the appendix and draws some conclusions from the exercise.

Diane Smith used a similar concept of a "completeness proof" [Smith 1972, chps. 3.7, 4.6, 5.7] to support another
analysis model for a different class of data structures (as discussed in Section 5.1.3). There, as here, the basic idea is to show that a model covers the necessary realm of data structures.

One of the first things which is clear from the modeling exercise in Appendix B is that some data structures must be modeled with more than one level of answers to a set of axes. Other data structures, which are in some sense "primitive" or "simple," need only one level of answers. An example of this latter case is the array which is modeled in Appendix B (and numerous other places throughout this thesis) as follows:

<table>
<thead>
<tr>
<th>Homogeneous</th>
<th>Basic items</th>
<th>Ordered</th>
<th>Number</th>
<th>Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

Here one set of answers from the aggregate model is sufficient. On the other hand the mathematical relation structure cannot be so easily conquered; it is not the same sort of primitive structure as the array. A "normalized" relation [Codd 1971] requires the following two levels:

<table>
<thead>
<tr>
<th>Relation Element</th>
<th>Homogeneous</th>
<th>Basic items</th>
<th>Ordered</th>
<th>Number</th>
<th>Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
<tr>
<td>Relation</td>
<td>Homogeneous</td>
<td>Basic items</td>
<td>Ordered</td>
<td>Number</td>
<td>Identification</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>FIXED</td>
<td>NAME, attribute name</td>
</tr>
</tbody>
</table>
As discussed previously in Section 3.2.2, this approach models the relation as a set of relation elements. In effect two primitive data structures, which when combined form a normalized relation, have been modeled separately. This procedure will be used throughout Appendix B when nonprimitive data structures are encountered. The same approach allows the modeling of several variations on a single data structure. For example, in Appendix B three different kinds of mathematical relations are modeled. The three all use the same first level but have different definitions of "relation element" at the second level.

With the above introduction, the reader may now fully appreciate Appendix B. Each main section uses one part of the data structure model of Section 3 to represent the common aggregates, associations, and files. This exercise clearly demonstrates the following:

1. The data structure model of this thesis can depict each of the data structuring techniques tabulated in Appendix A, and

2. Furthermore, the data structure model can discriminate among the possible variations and generalizations of individual data structuring techniques.

Thus the data structure model is complete; i.e., it is a capable model for common, contemporary data structures.

It is also interesting that Appendix B does not require the full power of the data structure model. There are some answers and some combinations of answers that do not appear at all. This demonstrates that the model applies to ad hoc data structures as well as the traditional ones modeled in Appendix B. Such data structures have been used throughout this thesis to describe the organization, decision table, and scheduling data base examples.
6.2 Usefulness for Top-Down Design

To demonstrate the worth of the top-down data structure design methodology (of Section 4) and to further investigate the use of the data structure model (of Section 3), this section uses both of them to attack a real world database design problem. The utility of the model and top-down method can be measured by seeing how well they perform on this practical data structure design.

The entire data structure design is presented in Appendix C along with the detailed requirements for the design. This section introduces the design problem in general terms and then examines just two representative portions of the full design. Finally, the appropriateness of the data structure model and top-down design for this task is discussed.

The example is drawn from the general area of large programming projects where a software system is produced from a number of individual programs representing the work of numerous programmers. For the purposes of the following discussion, these definitions hold:

Program: individual part of a software system; single compilation unit.

Version (of a program): the state of a program at a certain point in time.

Final product: the software system under development; the useful end product created.

Release (of a final product): an official, working collection of certain versions of some or all the programs; comprises the final product at a certain point in time.

Trouble report: official notification of trouble with the final product.
In terms of these definitions, the example data structure must satisfy the following requirements:

1. Numerous individual programs, each of which may have more than one version, must be stored. Each version has both an identifying version name and a chronological creation time.

2. Several different releases of the final product must be recorded. The information on each release includes the names and versions of the individual programs used and the date on which the release was produced.

3. A collection of trouble reports, their current status, and their relation to individual programs must be stored. For bugs which have been fixed, the trouble report notes which individual programs were changed and which release of the final product includes the corrections.

4. Numerous management statistics must also be available. The statistics include information on each individual program (size, last compile date, number of bugs fixed per version, etc.) and each release (number of bugs fixed, number of programs changed, etc.).

The specific data elements which implement these requirements are listed in Appendix C.

A version of a program incorporates and makes permanent some collection of changes, corrections, or revisions to the previous version. Thus versions exist along a single dimension time-wise and each version includes all the changes made to the previous one. Thus program text lines added and bugs fixed in one version remain in all subsequent versions unless they are explicitly modified. The relation between different releases is not as simple. It may be desirable to include some software features in one release but not in others (for example, to try out a new feature
while still allowing use of the current system). Thus releases are not linearly ordered in time; instead, some features will be in certain releases and not in others irrespective of the date of release. (Obviously, the same phenomenon applies to bugs.)

The only way releases can differ from each other is by containing different versions of the same programs or different programs. Thus a new release need not be created from the most recent version of each program. It is even probable that later releases may contain earlier versions of some program (for example, a program may operate so poorly in one release that in the next release an earlier, proven version of the program is used instead).

The requirements for the data structure design are complex; not only are there many kinds of information to be represented but the information is also interrelated in a complex fashion (as exemplified by the relationship between release and version). Nevertheless, these requirements are drawn from this author's extensive experience with practical software development projects and thus represent a common problem in large development efforts.

A general solution to this problem, as discussed here and detailed in Appendix C, is a contribution to the field of software engineering and program development. This design, as well as demonstrating the usefulness of this thesis, provides a solution to a real world data base problem. The design presented here will be useful to data base designers faced with similar design goals. The
creation and publishing of general solutions to common programming problems is seen by this author as a major step toward "software engineering" [Post 1970]. Another example of such a published general solution is [Hoare 1973a] which provides a generalized, parameterized software paging mechanism.

The specifics of the data structure design will not be discussed fully here for reasons of brevity (two fully described designs were presented in Section 4). The entire design, expressed pictorially, appears in Appendix C. This section next considers two key parts of the design which illustrate the advantages of the top-down data structure design method.

Level 0 of the design, as shown in Appendix C, deserves some attention since this is where someone using the design as documentation would begin. From level 0 alone it is immediately obvious that the data base contains two basic kinds of information: FINAL PRODUCT and TROUBLE REPORTS. This breakdown of the information is appropriate because trouble reports and the final product are the two major external features of the software development system. Level 0 also introduces and describes four abstractions. The two data structures depicted with the model and these four abstractions give a rather clear but still uncluttered view of the entire data base. The reader of the design would know that a FINAL PRODUCT is some general information (FP INFO) and descriptions of numerous releases (RELEASE INFO), one of which is distinguished as the LATEST RELEASE. Likewise, he/she would know that problems with the final product are represented as a set of individual TRs. Thus the data base has been introduced in a compact yet complete
way which sets the stage for the following levels. Both the compactness and completeness arise from the grouping decisions (see Section 4.4) made in the level 0 design.

We will also focus some attention here on levels 4 and 5 of the design (as shown in Appendix C). As of level 4 the data base has been relatively completely designed. The relationships between the final product, releases, programs, versions, and trouble reports has been fully designed by using names (PROGRAM NAME, RELEASE NAME, etc.) to represent the logical connections. The more complex relationships have made use of the PROG'S EDITION data structure (28) in a number of ways. The resulting structure, as of level 4, is suitable for implementation in a relational-style data management system. (PROG'S EDITION would in some cases result in unnormalized relations which might have to be normalized depending on the data base management system used.)

However, due to the lack of relational data management systems in most computer centers and the pointer's perennial popularity, it is worthwhile to continue the design to a more traditional implementation. Level 5 thus shows a redefinition decision (as defined in Section 4.4). This level (and level 6) replace the name-based connections between the elements of the data base with associations using the association model of Section 3.3. It is interesting to note that the association model is sufficient to represent all of the relational-style data organizations used at earlier levels including the unnormalized ones.
The point to be made here is not shown exclusively by either levels 4 or 5 but rather by the difference between them. There can be no denial of the relational model's conceptual clarity: levels 0 through 4 benefit from this clarity and the resulting design is easy to follow. However, for the reasons introduced above, the design must go further. The redefinition begun in level 5 allows drastic changes to be made and results in a more complex design (a capsule view of the final design is provided in level 6 by the restatement 0:6). This design, however, is easier to understand when it has been introduced via the earlier stages of the design. It seems obvious that the early, relational-style levels make the final, pointer-like design easier to understand. Further, a user of the data base might be content to view just the higher level, conceptual structure. Levels 0 to 4 provide a complete description of the information stored in the data base. A user who, for example, was using an existing management information system to query the data base need read no further than level 4. But still, all the details of the implementation are there for the users who want or require them.

The entire data base design (in Appendix C) demonstrates the usefulness of the methods used in at least two ways. First, the top-down data structure method provides a way of presenting the results in levels. This yields a design which is much more intellectually manageable. The reader should try to imagine the restatement shown in level 6 fleshed-out to show all the components of each data element. Such a design, probably appearing as a multiple fold-out page, could never be conquered and understood. On the other hand, the leveled design
presented here can be taken on a step at a time basis and the user can stop whenever he/she reaches a satisfactory level of detail.

Second, the design process, especially as aided by the redefinition feature, provides extremely useful documentation. The drawings of Appendix C are useful documentation just as they appear here. Thus the design process, due largely to its use of the data structure model to defined each component, provides sufficient documentation for the resulting data structures.

In conclusion it seems the model and top-down method provide just what was needed to design the software development data base. This success indicates that they may be fruitfully used on other real world data base design projects.

6.3 Ease of Use

It is claimed that the data structure model is readable, understandable, economic, and transparent, i.e., that the model is easy to use. Such a claim is, of course, subjective; however, it can be supported by comparing the data structure model of this thesis with other models. This section makes such a comparison based upon the same common example used in Section 5.1 (Figures 5-5 through 5-15).

The data structure to be considered here is shown in Figure 5-5 in terms of the data structure model. Reflecting first on this example by itself, two things are apparent:

1. The data structure model for this example is compact and economic, and
2. It presents the structural aspects of the example in a complete and understandable manner.

The pictorial nature of the data definition provides a clear organization for the facts to be presented. Reliance on shapes and the use of two dimensions eliminates the need for some prose and also makes it easier to "get the full picture" at a single glance. The data structure modeled in Figure 5-5 is quite complex (it is described at the end of Section 5.1); yet all aspects of the data structure are representable with the data structure model. The modeling process consists of going through the proper sections of the data structure model and treating each axis as a checklist, i.e., picking the proper answer for each question. This approach is quite easy to carry out because the model is fully described and the possible answers listed in Figures 3-4, 3-19, and 3-42. Thus the data structure model has performed quite well for this example.

Now the data structure model will be compared with the other models covered in Section 5.1 again with respect to the common example. This comparison will be carried out both in terms of individual models and the classification scheme for models (semantic, prototype, analysis, and information models, as introduced in Section 5.1).

Figures 5-6 through 5-15 express, as well as possible, the same example in each of the other models. The ability of each individual model to represent the example has been discussed in Sections 5.1.1, 5.1.2, 5.1.3, and 5.1.4. In comparison to the data structure model of this thesis the most serious lacks evidenced by Figures 5-6 through 5-15 are:

1. Overly large, verbose, or confusing representation of the data structure,
2. Incomplete modeling of all details of the example, and

3. Misleading representation of the intended structure of the example.

Of course each of these failings apply only to some of the other models and often with varying degrees of severity. The failings are also ameliorated by the fact that the models have widely different motivations and purposes.

Nevertheless, it seems clear that some definite comparisons can be made. Liskov and Zilles' implementation of Hoare's axiomatic model results in a very verbose modeling (point 1) of the common example, only a portion of which is shown in Figure 5-6. Shneiderman's Structured Data Structure (Figure 5-7) fails to model the SKILL FILE file structure (point 2) since it provides no way to represent selection via key values. Fleck's list structure (Figure 5-9) forces an unnecessary ordering on the components of PERSON and represents the "Cardinality" N-M HAS SKILL association in a way which, at best, can be described as nonobvious (point 3).

The transgressions of the other models on the intended structure of the example (points 2 and 3 above) are the most serious. This failure to represent true structure occurs in three major ways. First, many models force all the components of the example into a single style or form. This is true of Liskov and Zilles' axiomatic model, Earley's VERS, Fleck's list structures, Codd's relational model, Senko's Entity Set Model, and Backman's Data Structure Diagrams.
Second, many of the other models fail to represent some part of the example data structure. For example, some models provide a characterization for components of a data structure but no way to interrelate or connect separate elements. Specifically, this category includes Shneiderman's Structured Data Structures (no file structure, no "Cardinality" N-M association), Fleck's list structures (no file), Rosenberg's data graphs (total failure to represent practical structures), Earley's VERS2 (no "Cardinality" N-M association), Hsiao and Harary's generalized file (nothing but file structure), Senko's Entity Set Model (no "Cardinality" N-M association, no file), and Bachman's Data Structure Diagrams (no components, no file).

Third, many of the models force a particular implementation on the example structure. The following models all are based upon pointers or a pointer-like concept: Shneiderman's Structured Data Structures, Earley's VERS (links), Fleck's list structures, and Smith's Generalized Data Description Language (links and criteria).

The data structure model of this thesis suffers from none of these constraints. It covers three basically different kinds of data structuring techniques (aggregates, associations, and files) and a wide range of variations within each kind. By covering the three classes of data structures it is able to represent the wide variety of practical, real world data structures shown in Appendix A. Finally, by modeling the logical or conceptual organization of data structures, it avoids the trap of picking one particular implementation technique.

In all fairness, it must be pointed out that the data structure model stacks up so well because it was designed
with just those characteristics in mind. The other models (with the possible exception of the other analysis models) were often designed with significantly different objectives in mind. These objectives were summarized throughout the discussion of each individual model in Section 5. For example Fleck's list structures were developed to show a correspondence between a certain class of data structures and context free languages. In this case it is natural that the class of data structures needs to be somewhat restrictive and contrived.

In summary, the data structure model of Section 3 is easy to use for its intended purpose: defining the structural aspects of data structures. The model can be used to depict common, practical data structures in a way which, in comparison to existing models, is compact, economic, understandable and complete.
7. CONCLUSION

The preceding six sections and the appendixes have described a model of common data structures, shown how the model can be used for the top-down design of data structures, and compared and contrasted the model with related work. Section 7 concludes the reporting of this research by suggesting further work and noting the significant contributions of this thesis.

7.1 Further Work

The data structure model of this thesis allows the description of the structural aspects of data organizations; it motivates further work in four areas:

1. Further exploration of the data structure "space",
2. An analysis model for data access,
3. Further control over data integrity, and
4. An automatic programming implementation.

The data structure model defines three "spaces" of possible aggregates, associations, and files. Further examination of these spaces and their relationship to real world data structures may be enlightening. Section 3.2.2 presented a quick tour of the aggregate space and, in doing so, identified some practical data structures which are not offered by contemporary programming languages or data base management systems. Similar investigations of the association and file models and a comparison of the findings with the needs of programmer's would be worthwhile.

Such searching through data structure spaces is tedious. To aid these efforts at a better understanding of data
structures, it is interesting to speculate on a measure or "metric" over the space. This metric would attach a value to the distance or difference between any two data structures in the space, in effect, giving meaning to the innate concept of what makes data structures more or less different from each other. Than it would be possible to answer questions such as: Do the data structures in common use tend to cluster in one part of the space, as opposed to being evenly spread over the continuum?

At any rate, it seems clear from the numerous exercises carried out here that some answers to the data structure axes are more common than others. The question further work should answer is: Did today's popular data structures arise simply by chance and, if so, are there ignored, overlooked data structures of potential value?

Such further exploration using the data structure model is a direct extension of this thesis. The remaining topics to be discussed are suggested by the author's experiences in preparing this thesis but are not such direct continuations of this work.

A discussed previously in Sections 1.1 and 3.1, a conscious choice was made to model data structure independently from access considerations. The wisdom of this decision is clearly demonstrated by the success of the data structure model in representing common data structuring techniques. However, this experience does pave the way for an analysis-style data access model. Such work would complement the data structure model of this thesis by providing a similar model of contemporary access methods.
Existing work on understanding and modeling data access has tended to concentrate on a single data organization. Examples are [Carlson and Kaplan 1975] (relational model), [Reiter et al. 1972] (tree), and [Whitt and Sullenger 1975] (indexed sequential file). None of these can properly be called analysis models. Instead an analysis model for data access would cover a wide range of currently popular access methods. [Schaffner et al. 1972] presents a model which does cover some real world access methods. The model is not of an analysis nature but, instead, defines access techniques in terms of a collection of primitive macro functions. These macros operate on a graph-like model of the physical structure of files. However, this work may offer some insight into the question of general access models.

The structural model of this thesis may provide some insight into a possible form for such a data access model. Perhaps data access can be viewed as constrained by the answers to the data structure model. For example, considering the aggregate model, a "Homogeneous" axis answer of YES means a new element cannot be added to the aggregate unless it is of the same type as the current elements. Likewise, addition or deletion of elements must not contradict the current answer to the "Number" axis. Such an approach, perhaps introducing additional answers to the axes developed here, could probably model some access operations. However, it seems more likely that one or more entirely new sets of axes would provide surer control. The file model of this thesis, which perhaps infringes slightly on the realm of access, may offer some hints on the nature of these new axes.

As always, questions of data integrity seem to affect both data structure and access. The model of this thesis,
directed primarily to structural questions, achieved moderate success over some aspects of data integrity. The association model includes two axes solely for data integrity; however, it is still not able to represent certain kinds of constraints. A short discussion of a final example will indicate just how far the data structure model can be stretched and, at the same time, suggest direction for further work on data integrity.

The example will consider the data structure known as the B*-tree [Wedekind 1974; Bayer and McCreight 1972]. The essential details of this structure are defined as follows:

A **B-tree** is a tree such that

1. Each path from the root to any leaf has the same length \( h \),
2. Each node except for the root and the leaves has at least \( k+1 \) branches; the root is a leaf or has at least 2 branches,
3. Each node has at most \( 2k+1 \) branches,
4. Each node contains between \( k \) and \( 2k \) pairs consisting of a key and a datum.

A **B*-tree** is a B-tree such that

1. All the data are stored in leaf nodes; the non-leaf nodes contain only pointers and keys,
2. Non-leaf nodes contain between \( k^*+1 \) and \( 2k^*+1 \) pointers,
3. Leaf nodes contain one marker (to distinguish leaves) and between \( k \) and \( 2k \) key-datum pairs.

The structure thus constrains both the number of levels in the tree and the size of each level. The root of a B*-tree can have from 2 to \( 2k^*+1 \) pointers, each of which points to
another node. These nodes, if they are not leaves, each contain between \( k^*+1 \) and \( 2k^*+1 \) pointers. Numbering the levels of the tree \( 0, 1, \ldots, h \) (starting from the root), the number of nodes at the \( i \)th level, \( N_i \), is bounded as follows:

\[
2(k^*+1)^{i-1} \leq N_i \leq (2k^*+1)^i
\]

except at the top of the tree where there is exactly one node (the root). Since all paths through the tree are the same height, \( h \), the same formula limits the number of leaves:

\[
2(k^*+1)^{h-1} \leq N_h \leq (2k^*+1)^h
\]

These bounds are data integrity constraints of the highest order. If a B*-tree were modeled with the association model in the straightforward way used in Section 3.3 (each level as a "Cardinality" 1-N association), the constraints on the overall height of the tree and the size of each node could not be represented. The data integrity controls of the association model apply only to one association or one end data type at a time. There is no way to represent constraints over several association instances (as is required to limit the B*-tree's height). Similar lacks of the association model were discussed in Section 3.3.2 with regard to the example of Figure 3-40.

However, by being tricky, some of the constraints on B*-trees can still be represented with the data structure model. The secret is to begin by viewing the B*-tree as
a set of leaf nodes and setting the size of the set appropriately:

<table>
<thead>
<tr>
<th>B*-tree</th>
<th>Homogeneous:</th>
<th>YES, leaf nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Ordered:</td>
<td>YES, by key value</td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>Nh (as defined above)</td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Because the entire tree was introduced at once, one important constraint has been represented; now it remains to break the tree down into levels in the proper way. This requires both the proper number of nodes for each level and the proper number of pointers in each node. This process can be viewed as partitioning the set of leaves just introduced. Each level of the tree (working from the bottom-up) partitions the previous level. Each partition consists of $N_i$ sets (the number of nodes at the $i^{th}$ level) and the size of each set varies between $k^*+1$ and $2k^*+1$ (except for the top level which contains from 2 to $2k^*+1$ pointers in a single set). This method of building the tree from partitions can be modeled as follows:

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Homogeneous:</th>
<th>YES, partition set (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; i &lt; h-1</td>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Ordered:</td>
<td>YES, by key value</td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>$N_i$</td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Each level of the B*-tree is the proper number ($N_i$) of partition sets. The composition of a partition set likewise depends on the level. At the $h-1^{st}$ level, each partition set contains from $k^*+1$ to $2k^*+1$ leaves. This lowest level partition set can be modeled:

<table>
<thead>
<tr>
<th>Partition Set (h-1)</th>
<th>Homogeneous:</th>
<th>YES, leaf nodes of B*-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>$k^<em>+1 &lt; x &lt; 2k^</em>+1$</td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>
Since the original definition of $B^*$-tree specified $N_h$ leaves, it is guaranteed that $N_{h-1}$ such partition sets, each of the proper "Number" can be selected. Each remaining level must partition the previous level in a similar manner. Thus the elements of these partition sets are the partition sets of the next lower level, as follows:

<table>
<thead>
<tr>
<th>Partition Set (i)</th>
<th>Homogeneous: YES, partition set (i+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td>Number:</td>
<td>$k^<em>+1 \leq x \leq 2k^</em>+1$</td>
</tr>
<tr>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Finally, since the root has different bounds on its size, it may be represented:

<table>
<thead>
<tr>
<th>Partition Set (0)</th>
<th>Homogeneous: YES, partition set (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td>Number:</td>
<td>$2 \leq x \leq 2k^*+1$</td>
</tr>
<tr>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Because of the constraints on the number of leaves and the size and number of partition sets at each level, the 1st level will end up with between 2 and $2k^*+1$ elements to be connected to the root.

Thus, by adopting a rather unusual view of a tree and introducing the extension of a recursive data definition, the data integrity requirements of the $B^*$-tree can be fairly well modeled. The approach of starting with a tree as a set of leaves and then building up the levels may offer some new insight into the design and use of tree structures, but, in all fairness, it is rather obscure. The purpose of the data structure model is not to be obscure, but rather to make data structures easier to understand.
This experience with the B*-tree shows the limits of the data structure model: after a certain point data integrity can be controlled only by contorting the data structure. Further work on data integrity must provide a better method for handling global constraints which affect more than one level of a data structure. Such global data integrity constraints must be specified over the full data structure while still allowing detailed, piece-by-piece description of the individual components. Additionally, a more formal extension of the recursive data definition technique used here would be useful. The B*-tree seems an ideal example with which to test methods of representing data integrity constraints.

A third suggestion for extensions of this work concerns its implementation as an "automatic programming" system (see [Floyd 1972]). The data structure model is a nice, descriptive technique for describing the logical structure of data. Then, unfortunately, the user is "left hanging" as to how to implement the data structure. It seems that the axes and their answers could be used to automatically select ways of implementing every possible data structure. Earley suggested a similar idea, termed an "implementation facility" [Earley 1971], for the VERS language (discussed as a prototype model in Section 5.1.2). It would be interesting to see how much more difficult it is to add a similar facility to the analysis model of this thesis which covers a much richer range of data structures. The resulting automatic programming tool would provide a replacement for the numerous kinds of data declarations used in contemporary programming languages and data base management systems.
7.2 Contributions

The data structure model of Section 3 and the top-down design methodology of Section 4 are a contribution to our understanding and use of data structures. Specifically, this thesis is valuable for the following:

1. To teach data structures in a language-independent manner,

2. To choose and contrast data structures for practical programming tasks, and

3. To design and document data bases in an intelligible manner.

All these benefits arise from this work's success at modeling a wide class of common, real world data structures in a way which reveals their true substance.

Data structures can be taught in terms of the model instead of by a survey of individual programming languages. This approach would yield a better understanding of the true nature of data structures by showing how they vary and what their basic characteristics are. Appendix B is a guidebook for such an approach to data structures. A minimal set of classical data structures could be taught first, then others could be introduced as generalizations by changing certain axis answers as shown in the appendix. This teaching method could then be augmented by introducing the data structuring techniques of specific programming languages and seeing where they fit within the universe of possible data structures.
By its very style, the data structure model provides a concise way of both comparing existing data structures and selecting the features of new ones. When two existing data structures have been modeled, the axes for which their answers differ clearly indicate their variations. Similarly, by working through a set of axes as a checklist, the desired characteristics for a new data structure may be picked. In general, thinking about data structures has been moved to a higher plane, a level above that provided by individual names. The programmer or designer need no longer choose between PL/I arrays, PASCAL powersets, ELI self products, COBOL tables, SETL tuples, CODASYL DBTG sets, IDS group items, TDMS repeating groups, and so on, ad infinitum. Instead, a fixed set of relatively independent questions can be asked and answered one at a time.

The analysis model, by its very nature, facilitates a better understanding of the capabilities of and uses for data structures. Exercises such as the exploration of the universe of all possible aggregates in Section 3.2.2 can be easily carried out in terms of the n dimensions provided by each section of the data structure model. Such exercises allow a programmer to easily comprehend and become proficient with a large class of data structures.

Structured programming has concentrated attention on the intellectual manageability of the programming process. The data structure model of this thesis combined with the features of the top-down method for data structure design provide a way to design even the largest data bases in an intelligible, understandable manner. This design process produces complete documentation, also in a top-down style.

In addition, there are also some contributions which are incidental to the way the research has been carried out
and presented in this thesis. First, in order to describe the existing universe of data organization techniques, the data structures from 21 programming languages and data base management systems have been summarized in Appendix A. This summary provides a unique comparison of current programming tools. Second, a classification of data structure modeling techniques into four categories (semantic, prototype, analysis, and information models) has been introduced in Section 5.1. This classification is a contribution to the better understanding of the nature and purposes of data structure modeling. A single, common example has been expressed in terms of 11 different models to better show how each fits within the classification. Third, a rather sophisticated, practical data base for a software development system has been fully designed (Appendix C). This design is useful to others faced with similar requirements and, thus, represents a software engineering approach to data base design.

Much existing work in data base management and programming languages suggests a hard distinction between two or three levels of data definition. For example, it is common in data base systems to describe separate logical and physical levels; Earley's VERS2 (discussed in 5.1.2) proposes relational, access path, and machine levels. This thesis, and structured programming in general, urge a more merged multiple level view where the early levels are all logical or conceptual structure and the final level is entirely physical. The transitions between the two or more kinds of descriptions do not occur at any certain point but, rather, are a gradual process taking place over several levels of the design.

Each axis of the analysis model for common data structures portrays one basic characteristic of a class
of data structures. These axes represent the inherent concepts necessary to understand and distinguish between data structures. The lists of answers to each axis that have been enumerated here are sufficient to represent the current realm of data structures. When new data structuring capabilities evolve it may be necessary to extend the list of answers suggested here, but no new axes should be required. For example, the "Sequential" axis of the file model accepts only simple yes/no answers. In the future, files offering different kinds of sequential ordering may come into existence. One possibility is a file based upon a mathematical partial ordering instead of a strict linear sequence; such a change may be easily modeled by adding additional answers. By treating each list of possible answers in such an open-end fashion, the data structure model can continue to model all contemporary data structures.

The success of this thesis is due largely to the nature of the analysis style model. The approach of examining and analyzing what already exists has allowed this thesis to contribute both a better understanding of practical, contemporary data structures and a workable method for their top-down design.
APPENDIXES

A. Table of Data Structures in 21 Common Programming Languages and Data Base Management Systems

B. Completeness Exercise

C. Entire Software Development Data Base

D. Glossary of Terms
APPENDIX A

TABLE OF DATA STRUCTURES IN 21 COMMON PROGRAMMING LANGUAGES AND DATA BASE MANAGEMENT SYSTEMS

This appendix presents in tabular form the data structures provided by each of 21 programming languages and data base management systems (referred to with the generic term "system" hereafter). This table is one of the major inputs to the thesis and is in itself useful for comparing different systems.

The systems included were selected to be representative of both those in use today and those whose possibilities are currently being debated in the literature. Thus, numerous schools of thought from traditional to avant-garde are represented. The systems and references for each are listed along the top of the table. Each of the rows across the charts are numbered for ease of reading.

Each entry in the main body of the table describes the terms or statements used by a particular system to provide the data structure listed in the left-hand side headings. These headings were developed in the course of surveying the 21 systems. The four high order (leftmost) groupings or classes of data structures (i.e., basic items, aggregates, associations, and files) were adapted from the CODASYL Feature Analysis work [CODASYL 1971a]. The refinement from these general classes down to individual data structure names uses, whenever possible, names from the various systems being studied. Sometimes a second name is provided in parentheses following the first term to indicate two common names for a particular data structure. This thesis' discussion of these common data structures and development
of a data structure model are organized according to the left-hand table headings.

Obviously, when undertaking a classification exercise of this size, some questions arise. In some cases it is not patently obvious which row a system's feature should be listed in. Such features may be listed in two or more rows with appropriate comments. At any rate, the purpose of this thesis is not to suggest the left side of the table as the most pellucid names for the known data structures but rather to show a better way than names for understanding data structures.
<table>
<thead>
<tr>
<th>BASIC ITEMS</th>
<th>ALGOL 68 NAME FOR CONCEPT</th>
<th>BLISS NAME FOR CONCEPT</th>
<th>REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE</td>
<td>FULL WORD</td>
<td>BITS, BYTES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>PARTIAL WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MULTIPLE WORDS (FIXED LENGTH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINTER</td>
<td>GENERAL ADDRESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYPE QUALIFIED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYPELESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYPED (MODE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCodings</td>
<td>BUILT-IN ONLY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>USER DEFINED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUISITE NUMBERS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TABULAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OF N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BOOLEAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(TRUTH VALUE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PICTURE (PATTERN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONSTANT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPECIAL (DATE, TIME)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIXED OR MAX LENGTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARIABLE LENGTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLEX, STRING</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ELSEWHERE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>2 THINGS AT SAME TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 ALTERNATIVES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>ARRAY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATRIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N-TUPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEQUENCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATHEMATICAL RELATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N-ARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BINARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HIERARCHY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALTERNATIVES (OPTIONS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPEATING</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COUNT FIELD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AGGREGATES</td>
<td>TWO IDENTIFIERS FOR THE SAME OBJECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(GROUPS)</td>
<td>UNITED MODEL USING UNION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MULTIPLEXES</td>
<td></td>
<td>THE DEFAULT STRUCTURE</td>
</tr>
<tr>
<td>GROUP</td>
<td>TREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(STRUCTURE)</td>
<td>LINKED LIST</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIRECTED GRAPH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OWNER-MEMBER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSOCIATIONS</td>
<td>ABSTRACT ACCESS ALGORITHM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(RELATIONS)</td>
<td>KEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY BASIC ITEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPECIAL KEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CURRENT POINTERS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILES</td>
<td>DISTINGUISH DEFINITION AND INSTANCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEXING</td>
<td>TERM FOR DEFINITION</td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>TERM FOR INSTANCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECLARATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECLARATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>EL1</td>
<td>FORTRAN</td>
<td>MADCAP VI</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>[CODASYL 1973]</td>
<td>[WEGBREIT 1974]</td>
<td>[AMERICAN STANDARDS ASSOCIATION 1966]</td>
<td>[WELLS AND MORRIS 1973]</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>TYPE DEFINITION</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>LOGICAL</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>CONSTANT DEFINITION</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>EXPRESSION</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>CHAR, ALFA</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>USING</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>MAPPINGS</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>USING EQUIVALENCE</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>USING EQUIVALENCE</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>GENERAL TYPE</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>POWERSET, CLASS</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>SET</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>SEQUENCE WITH NUMBERED ELEMENTS</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>SEQUENCE WITH NUMBERED ELEMENTS</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td>USING CASE, THE DistinguishRED VARIABLES</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>43</td>
<td>DATA DIVISION</td>
<td>MODE</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>OBJECT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### XXXX Term in System Syntax

#### XXXX Name for Concept

<table>
<thead>
<tr>
<th>Storage</th>
<th>Partial Word</th>
<th>Multiple Words [Fixed Length]</th>
<th>General Address</th>
<th>Type Qualified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Word</td>
<td>User Defined</td>
<td>Regular Numbers [Tabular</td>
<td>Boolean Truth Value</td>
<td>Picture Pattern</td>
</tr>
<tr>
<td>Pointer</td>
<td>Built In Only</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Basic Items

<table>
<thead>
<tr>
<th>Conceptual Item</th>
<th>Encodings</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Numbers</td>
<td>Tabular</td>
<td>2 Things at Same Time</td>
</tr>
<tr>
<td>Boolean Truth Value</td>
<td>Picture Pattern</td>
<td>2 Alternatives</td>
</tr>
<tr>
<td>Numeric and String Variables</td>
<td>Using Preprocessor</td>
<td>Defined Position</td>
</tr>
<tr>
<td>Constant</td>
<td>Special</td>
<td>2 Things at Same Time</td>
</tr>
<tr>
<td>Date Time</td>
<td>Variable Length</td>
<td>2 Alternatives</td>
</tr>
<tr>
<td>Strings</td>
<td>Virtual</td>
<td>Defined Position</td>
</tr>
</tbody>
</table>

#### Aggregates (Groups)

<table>
<thead>
<tr>
<th>Table</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Matrix</td>
</tr>
<tr>
<td>Set</td>
<td>Tuple</td>
</tr>
<tr>
<td>n-Tuple</td>
<td>Sequence</td>
</tr>
<tr>
<td>Sequence</td>
<td>Mathematical Relation</td>
</tr>
<tr>
<td>N-Ary</td>
<td>Binary</td>
</tr>
</tbody>
</table>

#### Group (Structure)

<table>
<thead>
<tr>
<th>Group (Structure)</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchy</td>
<td>Alternatives</td>
</tr>
<tr>
<td>Alternatives</td>
<td>Options</td>
</tr>
</tbody>
</table>

#### Pointer Based

<table>
<thead>
<tr>
<th>Pointer Based</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>Directed Graph</td>
</tr>
</tbody>
</table>

#### Other Associations

<table>
<thead>
<tr>
<th>Other Associations</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner-Member</td>
<td>Functional</td>
</tr>
</tbody>
</table>

#### Arbitrary Access Algorithm

<table>
<thead>
<tr>
<th>Arbitrary Access Algorithm</th>
<th>Keyed or Direct Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>Keyed or Direct Files</td>
</tr>
<tr>
<td>Key</td>
<td>Keyed or Direct Files</td>
</tr>
<tr>
<td>Special Key</td>
<td>Keyed or Direct Files</td>
</tr>
<tr>
<td>Current Pointers</td>
<td>Keyed or Direct Files</td>
</tr>
<tr>
<td>Sequential</td>
<td>Keyed or Direct Files</td>
</tr>
</tbody>
</table>

#### Distinguish Definition and Instance?

<table>
<thead>
<tr>
<th>Distinguish Definition and Instance?</th>
<th>Term for Definition</th>
<th>Term for Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

---

- 368 -
<table>
<thead>
<tr>
<th>SIMULA</th>
<th>TYPICAL ASSEMBLY LANGUAGE</th>
<th>VDL</th>
<th>VERBZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DAHL ET AL. 1972]</td>
<td>[CMBIAN AND MOORE 1972]</td>
<td>[WEINER 1972]</td>
<td>[BARLEY 1973]</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>2</td>
<td>YES</td>
<td>2</td>
<td>YES</td>
</tr>
<tr>
<td>3</td>
<td>YES</td>
<td>3</td>
<td>YES</td>
</tr>
<tr>
<td>4</td>
<td>YES</td>
<td>4</td>
<td>YES</td>
</tr>
<tr>
<td>5</td>
<td>REF TO OBJECTS ONLY</td>
<td>5</td>
<td>YES</td>
</tr>
<tr>
<td>6</td>
<td>TEXT</td>
<td>6</td>
<td>YES</td>
</tr>
<tr>
<td>7</td>
<td>CLASS</td>
<td>7</td>
<td>YES</td>
</tr>
<tr>
<td>8</td>
<td>INTEGER, REAL</td>
<td>8</td>
<td>ELEMENTARY OBJECTS INT. REAL</td>
</tr>
<tr>
<td>9</td>
<td>BOOLEAN</td>
<td>9</td>
<td>YES</td>
</tr>
<tr>
<td>10</td>
<td>BOOLEAN</td>
<td>10</td>
<td>YES</td>
</tr>
<tr>
<td>11</td>
<td>SELECTORS MAY BE</td>
<td>11</td>
<td>YES</td>
</tr>
<tr>
<td>12</td>
<td>ELEMENTARY OBJECTS</td>
<td>12</td>
<td>YES</td>
</tr>
<tr>
<td>13</td>
<td>CHARACTER, TEXT</td>
<td>13</td>
<td>STRING</td>
</tr>
<tr>
<td>14</td>
<td>ELEMENTARY OBJECTS</td>
<td>14</td>
<td>YES</td>
</tr>
<tr>
<td>15</td>
<td>VIRTUAL ENTITIES</td>
<td>15</td>
<td>YES</td>
</tr>
<tr>
<td>16</td>
<td>ARRAY</td>
<td>16</td>
<td>YES</td>
</tr>
<tr>
<td>17</td>
<td>COMPOSITE OBJECTS USING</td>
<td>17</td>
<td>YES</td>
</tr>
<tr>
<td>18</td>
<td>ELEM(i) SELECTORS</td>
<td>18</td>
<td>YES</td>
</tr>
<tr>
<td>19</td>
<td>SEQUENCE USING</td>
<td>19</td>
<td>YES</td>
</tr>
<tr>
<td>20</td>
<td>ELEMENT NUMBERS</td>
<td>20</td>
<td>YES</td>
</tr>
<tr>
<td>21</td>
<td>ARRAY</td>
<td>21</td>
<td>YES</td>
</tr>
<tr>
<td>22</td>
<td>COMPOSITE OBJECTS</td>
<td>22</td>
<td>YES</td>
</tr>
<tr>
<td>23</td>
<td>USING ELEM(i) SELECTORS</td>
<td>23</td>
<td>YES</td>
</tr>
<tr>
<td>24</td>
<td>SEQUENCE USING</td>
<td>24</td>
<td>YES</td>
</tr>
<tr>
<td>25</td>
<td>ELEMENT NUMBERS</td>
<td>25</td>
<td>YES</td>
</tr>
<tr>
<td>26</td>
<td>COMPOSITE OBJECTS</td>
<td>26</td>
<td>YES</td>
</tr>
<tr>
<td>27</td>
<td>USING ELEM(i) SELECTORS</td>
<td>27</td>
<td>YES</td>
</tr>
<tr>
<td>28</td>
<td>SEQUENCE USING</td>
<td>28</td>
<td>YES</td>
</tr>
<tr>
<td>29</td>
<td>ELEMENT NUMBERS</td>
<td>29</td>
<td>YES</td>
</tr>
<tr>
<td>30</td>
<td>COMPOSITE OBJECTS</td>
<td>30</td>
<td>YES</td>
</tr>
<tr>
<td>31</td>
<td>USING ELEM(i) SELECTORS</td>
<td>31</td>
<td>YES</td>
</tr>
<tr>
<td>32</td>
<td>SEQUENCE USING</td>
<td>32</td>
<td>YES</td>
</tr>
<tr>
<td>33</td>
<td>ELEMENT NUMBERS</td>
<td>33</td>
<td>YES</td>
</tr>
<tr>
<td>34</td>
<td>COMPOSITE OBJECTS</td>
<td>34</td>
<td>YES</td>
</tr>
<tr>
<td>35</td>
<td>USING ELEM(i) SELECTORS</td>
<td>35</td>
<td>YES</td>
</tr>
<tr>
<td>36</td>
<td>SEQUENCE USING</td>
<td>36</td>
<td>YES</td>
</tr>
<tr>
<td>37</td>
<td>ELEMENT NUMBERS</td>
<td>37</td>
<td>YES</td>
</tr>
<tr>
<td>38</td>
<td>COMPOSITE OBJECTS</td>
<td>38</td>
<td>YES</td>
</tr>
<tr>
<td>39</td>
<td>USING ELEM(i) SELECTORS</td>
<td>39</td>
<td>YES</td>
</tr>
<tr>
<td>40</td>
<td>SEQUENCE USING</td>
<td>40</td>
<td>YES</td>
</tr>
<tr>
<td>41</td>
<td>ELEMENT NUMBERS</td>
<td>41</td>
<td>YES</td>
</tr>
</tbody>
</table>

- 369 -
<table>
<thead>
<tr>
<th>XXXX TERM IN SYSTEM SYNTAX</th>
<th>XXXX NAME FOR CONCEPT</th>
<th>REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE</td>
<td>FULL WORD</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>PARTIAL WORD</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>MULTIPLE WORDS</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(FIXED LENGTH)</td>
<td></td>
</tr>
<tr>
<td>POINTER</td>
<td>GENERAL ADDRESS</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>TYPE QUALIFIED</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>TYPELESS</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>TYPED (MODE)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>BUILT-IN ONLY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USER DEFINED</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>IN A LIMITED WAY AS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IMPLEMENTOR TYPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REGULAR NUMBERS</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TABULAR (1 OF N)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>CHECK IN RANGE OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>THRU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BOOLEAN (TRUE VALUE)</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>BIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PICTURE (PATTERN)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>PICTURE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONSTANT</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>SPECIAL (DATE, TIME)</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>DATA BASE KEY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRINGS</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>FIXED OR MAX. LENGTH</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>YES, LENGTH NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KNOWN TO USER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USING OCCURRING TO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPECIFY LENGTH FIELD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VIRTUAL RECORD OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VIRTUAL SOURCE</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>EB OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EQUIVALENTS</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2 AT THE SAME TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 ALTERNATIVES</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>PECKER, OCCURRING ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMER</td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>ARRAY</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>MATRIX</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>SET</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>N-TUPLE</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>SEQUENCE</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>MATHEMATICAL RELATION</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>N-ARY</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>FIRST NORMAL FORM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RELATIONS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HIERARCHY</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>ALTERNATIVES (OPTIONS)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPEATING</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>COUNT FIELD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>DATA AGGREGATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCCURRING COUNT FIELD</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>(TIMERS)</td>
<td></td>
</tr>
<tr>
<td>AGGREGATES (GROUPS)</td>
<td>TREE</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>LINKED LIST</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>DIRECTED GRAPH</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>OWNER-MEMBER</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSOCIATIONS (RELATIONS)</td>
<td>TREE</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>LINKED LIST</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>DIRECTED GRAPH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OWNER-MEMBER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHER ASSOCIATIONS</td>
<td>TREE</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>LINKED LIST</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>DIRECTED GRAPH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OWNER-MEMBER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td></td>
</tr>
<tr>
<td>ARBITRARY ACCESS ALGORITHM</td>
<td>TREE</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>LINKED LIST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIRECTED GRAPH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OWNER-MEMBER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL</td>
<td></td>
</tr>
<tr>
<td>FILES</td>
<td>ARBITRARY ACCESS ALGORITHM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INDEXING</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY BASIC ITEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPECIAL KEY</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>CURRENT POINTERS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
</tr>
<tr>
<td>DISTINGUISH DEFINITION AND</td>
<td>YES</td>
<td>43</td>
</tr>
<tr>
<td>INSTANCES</td>
<td>TERM FOR DEFINITION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TERM FOR INSTANCE</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>SCHEMA OR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA MODEL DEFINITION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SCHEMA, SUBSCHEMA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TUPLE VARIABLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCCURRENCE</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>---------------------------</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section 6.1 introduces the idea of completeness for a data structure model. This appendix demonstrates the completeness of the data structure model of this thesis by expressing each of the nonbasic item data structures from Appendix A in terms of the model.

The following exercise is divided into three sections, one each for the aggregate, association, and file data structures. Each data structure is modeled using the appropriate part of the data structure model. The name of each data structure appears at the left margin followed by two pieces of cross reference information: the row of Appendix A for the data structure (e.g., A 21) and a figure number from Section 2 showing an example of the data structure (e.g., 2-22). Each data structure is modeled using the abbreviated form of the axis questions as introduced in Figures 3-4 (aggregates), 3-19 (associations), and 3-42 (files).

As pointed out in the introduction to Appendix A and throughout Section 2, many variations and extensions have been introduced for the more widely used data structures. The following exercise attempts to first model each data structure in its original, most restrictive, or classical sense. Where possible the particular characteristics which usually identify or distinguish one data structure from others is noted. These characteristics are the ones which are most forcefully associated with the data structure name. Next, possible extensions and generalizations of the data structure are mentioned and their effect on the modeling
process noted. This approach seems to present, in a somewhat orderly fashion, the evolution and growth of both individuals and classes of data structures.

**AGGREGATES**

<table>
<thead>
<tr>
<th>Array</th>
<th>Homogeneous:</th>
<th>Basic items:</th>
<th>Ordered:</th>
<th>Number:</th>
<th>Identification:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 21</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
<tr>
<td>2-22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This sort of array is offered by many languages including FORTRAN. Less restrictive kinds of arrays also exist. Many languages (VERS2, PASCAL, EL1, etc.) provide arrays with "Basic items" answered NO. It is also common to allow "Number" to be either LIMITED or UNBOUNDED (ALGOL 68, COBOL, CODASYL DBTG). The name array is most closely associated with "Homogeneous" and "Ordered" answered YES and "Identification" NUMBER. The less restrictive kinds of arrays rapidly become confused with n-tuples and sequences.

| Matrix    | Homogeneous: | Basic items: | Ordered: | Number:                 | Identification: |
|-----------|--------------|--------------|----------|-------------------------|
| A 22      | YES          | YES          | YES      | FIXED, n-tuple          | NUMBER         |
| 2-25      |              |              |          |                         |                |

The matrix is very similar to the array; an ordered n-tuple of numbers is used for identification instead of a single number. The same generalizations and confusions which apply to the array also affect the matrix.

<table>
<thead>
<tr>
<th>Set</th>
<th>Homogeneous:</th>
<th>Basic items:</th>
<th>Ordered:</th>
<th>Number:</th>
<th>Identification:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 23</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>UNBOUNDED</td>
<td>NONE</td>
</tr>
<tr>
<td>2-28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A set is characterized by an "Ordered" answer of NO; all the other axes take on different answers in various systems. Some versions of set structures need not be homogeneous, may consist only of basic items, or may have LIMITED or FIXED "Number." "Identification" may be answered either NONE or POINTER; an answer of NAME implies a hierarchy structure and an answer of NUMBER is not provided by any of the systems in Appendix A. Numerous versions of set structures have been discussed in Section 3.2.

<table>
<thead>
<tr>
<th>N-tuple</th>
<th>Homogeneous:</th>
<th>Basic items:</th>
<th>Ordered:</th>
<th>Number:</th>
<th>Identification:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 24</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>FIXED</td>
<td>NUMBER</td>
</tr>
<tr>
<td>2-29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The n-tuple can be most clearly viewed as a generalization of the array; it allows nonhomogeneous elements to be grouped together in an ordered fashion and identified by indexes. Some languages (e.g., MADCAP VI) provide n-tuples of other than basic items.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Homogeneous:</th>
<th>Basic items:</th>
<th>Ordered:</th>
<th>Number:</th>
<th>Identification:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 25</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>UNBOUNDDED</td>
<td>NONE</td>
</tr>
<tr>
<td>2-31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sequence is also obviously related to the array. It can be viewed either as a generalization allowing UNBOUNDDED "Number" or as a more primitive structure which simply orders its elements and does not provide any means of identification. Sequences may sometimes be restricted to "Basic items" YES.
The aggregates considered so far in this appendix have all been primitive data structures; the mathematical relation is here shown to be a combination of two different structures. At the highest level, all versions of the relation may be modeled alike. The differences between versions arise at the second level. Here three possible second level data definitions are used to show the variations possible. This particular example is discussed further within Section 6.1.

The hierarchy is another basic data structure provided by many systems (14 of the 21 in Appendix A). Its major distinguishing characteristic is the use of NAMES to identify components. These components need not be all alike nor are they restricted to basic items. "Number"
is usually FIXED although it is conceivable to have UNBOUNDED "Number" by allowing names to be picked at will from some general pattern or type. (The VERS language, as discussed in Section 5.1.2, includes such a feature. Data structures using this sort of feature are shown in Appendix C.) The only other possible variation from the above axis answers is to order the elements according to the order in which their names are specified. This ordering is rare (PL/I is one example). Figures 2-35 (one level hierarchy) and 2-36 (nested hierarchy) are both the same data structuring technique. Appendix A also lists the hierarchy with alternatives (as shown in Figure 2-38). This data structure can be modeled as an equivalence of alternatives (row 20 in Appendix A) between each of the alternative hierarchies.

<table>
<thead>
<tr>
<th>Repeating</th>
<th>Homogeneous:</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 30</td>
<td>Basic items:</td>
<td>NO</td>
</tr>
<tr>
<td>2-39</td>
<td>Ordered:</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>Number:</td>
<td>UNBOUNDED</td>
</tr>
<tr>
<td></td>
<td>Identification:</td>
<td>NONE</td>
</tr>
</tbody>
</table>

As pointed out in Section 3.2.2, repeating structures and sets are very similar. The same generalizations and variations that apply to set also apply to repeating. The "count field" of a repeating structure (Appendix A, line 31) is just a semantic connection between a repeating structure and a regular number basic item. This connection could be represented with the data structure model using is.

ASSOCIATIONS

("Exclusive" axis answers are usually omitted because this axis applies only to data structures built-up from more than a single association and also depends upon the semantics of the data structure.)
Tree data structures vary widely in common use; different answers to the association model's questions may be needed at each level of the tree. The most common structure at each level is the one shown here. The distinguishing characteristics of the tree are "Cardinality" 1-N and "Complete" YES,YES. The other answers may vary. "Kinds of ends" 1,N is possible (Figure 2-44) as is "Loop" YES (Figure 3-25).

Linked List

Cardinality: 1-1
Kinds of ends: 1,1
Loop: YES
Complete: YES,YES

The basic form of linked list, modeled as shown, may be extended in several ways. For example, headcell links may be added as a separate association:

headcell link
Cardinality: 1-N (one headcell)
Kinds of ends: 1,1
Loop: NO
Complete: YES,YES

However the model cannot distinguish between forward and backward links because associations are undirected. The distinguishing characteristics of a linked list are "Cardinality" 1-1 and "Loop" YES. Additional associations with "Loop" NO and "Cardinality" 1-1 may be used at the beginning and end of the list if a headcell or tailcell are present. Generalizations such as these and the headcell link shown above, are introduced with separate data structures which can be modeled independently, as opposed to changing the answers to the original linked list model.
Directed graphs are relatively free form associations among a set of end data structures. "Cardinality" N-M implies that every end element may be connected to any number of other elements and "Complete" NO,NO means that these connections are all optional. This sort of directed graph, and the example shown in Figure 2-46, assume the component elements are all of one kind. A generalization which both MADCAP VI and IMS provide allows more than one kind of element to be associated in a directed graph. This generalization is modeled by changing "Kinds of ends" to x,x where x is the number of kinds of elements.

Owner-Member | Cardinality: 1-N (one owner) |
---|---|
A 35 | Kinds of ends: 1,x x ≥ 1 |
2-47 | Loop: NO |
| Complete: YES,YES |

The owner-member structure as implemented in CODASYL DBTG has been discussed extensively in Section 3.3.2. Its basic characteristics are "Cardinality" 1-N and "Kinds of ends" 1,x. Although it seems to be a needless constraint, both CODASYL DBTG and IDS require that "Loop" be NO, thus not allowing the same kind of element to be both owner and member. IDS requires the "Complete" YES,YES answer shown above since every owner element must own a (possibly empty) set of members and each member must belong to some owner. CODASYL DBTG allows both owners and members to exist independently; this generalization is modeled with a "Complete" answer of NO,NO.
The functional association is essentially a complete mathematical function; thus, each domain element at its A-end is associated with a single range element at its B-end. The model shown presents this function in its most general form. Various restrictions of a mathematical nature can also be represented with the model. For example, a one-to-one function has "Cardinality" 1-1 and a onto function is "Complete" YES,YES.

FILES

Arbitrary Access Algorithm
A 37

The arbitrary access algorithm cannot be modeled in any general form due to its basic characteristic of being completely arbitrary. Any particular data organization implemented as an arbitrary access algorithm can be modeled with the appropriate section of the data structure model. For example, if an array is the desired data structure it is still modeled as shown at the beginning of this exercise.

Key File  Selection:  BASIC ITEM KEY
A 38     Unique:  YES
2-49     Sequential:  NO
Kinds of entries:  1

The key file is identified by "Selection" BASIC ITEM KEY. If each key value specifies a single entry then "Unique" is YES as shown. A common extension is to allow "Unique" NO in which case each key value may select any number of entries. A less common generalization is to allow
a key value to select entries of two or more different kinds ("Kinds of entries" ≥ 2).

<table>
<thead>
<tr>
<th>Special Key</th>
<th>Selection:</th>
<th>File</th>
<th>Unique:</th>
<th>A 39</th>
<th>Sequential:</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPECIAL KEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special Key</th>
<th>Selection:</th>
<th>Current Pointer</th>
<th>Unique:</th>
<th>A 40</th>
<th>Sequential:</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Currentness</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special key files are like key files except the key values are specially created by the system to identify entries. A characteristic of these key values is that "Unique" is always YES. Again the file may be generalized to contain more than one kind of entry; in this case the special key value selects a unique entry from among all the different kinds present.

<table>
<thead>
<tr>
<th>Current Pointer</th>
<th>Selection:</th>
<th>Sequential</th>
<th>Unique:</th>
<th>A 40</th>
<th>Sequential:</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CURRENTNESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current pointer files also uniquely identify entries based upon the concept of currentness. With the data structure model, one "Selection" CURRENTNESS file is used for each current pointer available to the user. The only generalization is again to "Kinds of entries" greater than 1.

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Selection:</th>
<th>Unique:</th>
<th>Sequential</th>
<th>Unique:</th>
<th>A 41</th>
<th>Sequential:</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NONE</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A sequential file is characterized by the "Sequential" YES answer. Only access according to a strict sequential order is provided. "Kinds of entries" may again be extended to allow different file elements.
Perhaps the most common file generalization is to combine two or more of the file types mentioned above. For example the popular "indexed sequential" file is a melding of the strictly sequential file with the key file. In such cases the answers to the modeling questions are also combined, picking the most general answer for each axis.
APPENDIX C

ENTIRE SOFTWARE DEVELOPMENT DATA BASE

Section 6.2 argues the usefulness and practicality of this thesis' data structure model and top-down method by appealing to an example which is described in its entirety in this appendix. The example data base is described in general terms in Section 6.2. This appendix begins by listing its individual details and then presents a top-down design of the entire data base.

The software development data base contains information on individual programs and the ways in which they are combined and released as a final product. For each individual program, the following details must be recorded (the curly brackets indicate multiple occurrences):

Program:

name
(current versions)
latest version
date and time of latest compilation
cumulative changes
   a) for bugs corrected
   b) for enhancements
total number of versions which have ever existed
description (i.e., program's purpose or function)

Thus a program has a name and exists in different chronological versions; some number of recent versions are considered "current" and kept in the data base. This number may be less than the total number of versions which have ever been created during the life of a program.
New versions of a program are created to incorporate changes, improvements, and corrections to the final product. For each current version, the following information is stored:

Version:
- version name
- date and time of creation
- size (lines of code, etc.)
- number of changes
  - a) for bugs corrected
  - b) for enhancements
- description

Versions of programs are grouped together periodically and released to the users of the final product. Numerous "releases" may be in use at any given time. Thus, information on both the final product in general and releases of it must be kept, as follows:

Final Product:
- {current releases}
- latest release
- total number of releases which have ever existed
- description

Release:
- release name
- date and time of creation
- {programs x versions used}
- number of changes to programs
- number of bugs fixed
- number of new program versions since last release
- description

So at any given time the data base contains the description of any number of releases of the final product. Naturally, these releases will have some bugs; as problems are reported by the users, "trouble reports" are also added to the data base. Each trouble report specifies:
Trouble Report:

- date and time received
- status: open, ready for release, or completed
- description (of the problem)

if status = ready for release
    {programs x versions changed}
if status = completed
    {programs x versions changed}
    release incorporating fix

The remainder of this appendix presents the top-down description of a data structure which meets these requirements. Some parts of the design are discussed in Section 6.2, but for the most part the design is simply presented in its final form. This approach allows investigation of the claim that designs using the data structure model and top-down method of this thesis provide useful documentation for data structures.
FP INFO is required data on the entire final product as opposed to facts about any single release of it.

LATEST RELEASE is identification for most recent release of final product.

RELEASE INFO is description of a single release of the final product; one occurrence exists for each single release which is "current"; i.e., is still maintained in the data base.

TR is a single trouble report in any possible status.

LEVEL 0
1A IS DEFINED

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED, 2
NUMBER OF PROGRAMS IN
RELEASE
IDENTIFICATION: NAME,
INCLUDING ANY PROGRAM

RELEASE INFO

RELEASE NAME: 4A
RELEASE DESC: 3B
[PROGRAM NAME]
PROG'S EDITION: 2B, 6A

LEVEL 1 PART 1

1B IS DEFINED

RELEASE NAME

LATEST
RELEASE

4A
PROGRAM IS DEFINED

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 2
IDENTIFICATION: NAME

PROGRAM FILE

PROGRAM NAME

VERSION IS DEFINED

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 3
IDENTIFICATION: NAME

VERSION NAME

CURRENT VERSIONS

C= YES
N

SELECTION: BASIC ITEM KEY,
PROGRAM NAME

UNIQUE: YES
SEQUENTIAL: NO

LEVEL 1 PART 2
RELEASE NAME IS IDENTIFYING NAME FOR A CURRENT VERSION OF THE FINAL PRODUCT; THE NAME IS UNIQUE OVER THE ENTIRE DATA BASE.

RELEASE DESC IS A DESCRIPTION OF ADDITIONAL RELEASE SPECIFIC INFORMATION.

PROG'S EDITION IS A PARTICULAR VERSION OF A PARTICULAR PROGRAM; ONE OCCURRENCE FOR EACH PROGRAM USED IN THIS RELEASE.

PROGRAM NAME IS IDENTIFYING NAME FOR A PROGRAM; THE NAME IS UNIQUE OVER THE ENTIRE DATA BASE.

PROGRAM INFO IS OTHER INFORMATION ABOUT ONE PROGRAM.

VERSION NAME IS IDENTIFYING NAME FOR A VERSION OF A PROGRAM; THE NAME IS UNIQUE WITHIN THE VERSIONS OF A SINGLE PROGRAM.

DATETIME IS USED HERE AS TIME OF CREATION OF A VERSION.

VERSION INFO IS OTHER INFORMATION ABOUT ONE PROGRAM.

LEVEL 1 PART 3
2: 2A IS DEFINED

EQUIVALENCE - ALTERNATIVES

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 3
IDENTIFICATION: NAME

RECEIVED | STATUS | DESCRIPTION
---------|--------|-------------
DATE-TIME | CONSTANT STRING
1 IS OPEN

4D

TR

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: UNBOUNDED, 3 + NO. OF AFFECTED PROGRAMS
IDENTIFICATION: NAME

RECEIVED | STATUS | AFFECTS | DESCRIPTION
---------|--------|---------|-------------
DATE-TIME | CONSTANT 2 IS READY | PROG'S EDITION

4D

2B, 6A

LEVEL 2 PART 1
2B IS DEFINED

PROG'S EDITION

N-TUPLE

HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: YES
NUMBER: FIXED, 2
IDENTIFICATION: NUMBER

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PROGRAM NAME</td>
<td>VERSION NAME</td>
</tr>
<tr>
<td>4B</td>
<td>4C</td>
</tr>
</tbody>
</table>

0:2 FINAL PRODUCT IS INFORMATION ABOUT THE FINAL PRODUCT IN GENERAL AND ABOUT ANY NUMBER OF RELEASES, ONE OF WHICH IS THE MOST RECENT ONE. EACH RELEASE IS REPRESENTED AS SOME GENERAL INFORMATION AND A SET OF PROGRAM NAME-VERSION NAME PAIRS FOR EACH VERSION OF A PROGRAM USED IN THE FINAL PRODUCT. EACH PROGRAM AND EACH OF ITS CURRENT VERSIONS ARE DESCRIBED INDIVIDUALLY.

TROUBLE REPORTS IS A SET OF INFORMATION ON EACH TROUBLE REPORT—THIS INFORMATION TAKES ONE OF THREE POSSIBLE FORMS.

LEVEL 2 PART 2
### 3A is Defined

<table>
<thead>
<tr>
<th>HIERARCHY</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOMOGENEOUS: NO</td>
</tr>
<tr>
<td>BASIC ITEMS: YES</td>
</tr>
<tr>
<td>ORDERED: NO</td>
</tr>
<tr>
<td>NUMBER: FIXED, 2</td>
</tr>
<tr>
<td>IDENTIFICATION: NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TOTAL RELEASES</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>STRING</td>
</tr>
</tbody>
</table>

### 3B is Defined

<table>
<thead>
<tr>
<th>HIERARCHY</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOMOGENEOUS: NO</td>
</tr>
<tr>
<td>BASIC ITEMS: NO</td>
</tr>
<tr>
<td>ORDERED: NO</td>
</tr>
<tr>
<td>NUMBER: FIXED, 5</td>
</tr>
<tr>
<td>IDENTIFICATION: NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RELEASE DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATION NUMBER</td>
</tr>
<tr>
<td>CHANGES FIXED</td>
</tr>
<tr>
<td>NEW VERSIONS</td>
</tr>
<tr>
<td>DESCRIPTION STRING</td>
</tr>
</tbody>
</table>

### 3D is Defined

<table>
<thead>
<tr>
<th>HIERARCHY</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOMOGENEOUS: NO</td>
</tr>
<tr>
<td>BASIC ITEMS: NO</td>
</tr>
<tr>
<td>ORDERED: NO</td>
</tr>
<tr>
<td>NUMBER: FIXED, 3</td>
</tr>
<tr>
<td>IDENTIFICATION: NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VERSION INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
</tr>
<tr>
<td>CHANGES</td>
</tr>
<tr>
<td>DESCRIPTION STRING</td>
</tr>
</tbody>
</table>

### 3C is Defined

<table>
<thead>
<tr>
<th>HIERARCHY</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOMOGENEOUS: NO</td>
</tr>
<tr>
<td>BASIC ITEMS: NO</td>
</tr>
<tr>
<td>ORDERED: NO</td>
</tr>
<tr>
<td>NUMBER: FIXED, 5</td>
</tr>
<tr>
<td>IDENTIFICATION: NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LATEST VERSION</th>
<th>LAST COMPILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION NAME</td>
<td>DATE-TIME</td>
</tr>
<tr>
<td>CHANGES</td>
<td>CHANGE NUM</td>
</tr>
<tr>
<td>TOTAL VERSIONS</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DESCRIPTION STRING</td>
<td></td>
</tr>
</tbody>
</table>

LEVEL 3
4A IS DEFINED

PICTURE

RELEASE NAME: AAAAA 'U'AAAAA
E.G. SWAP_PAG

4B IS DEFINED

PICTURE

PROGRAM NAME: AAAAA
E.G. ASCAN

4C IS DEFINED

PICTURE

VERSION NAME: 'V'99V99
E.G. V5.2

4D IS DEFINED

HIERARCHY

HOMOGENEOUS: NO
BASIC ITEMS: YES
ORDERED: NO
NUMBER: FIXED, 6
IDENTIFICATION: NAME

DATE TIME

MONTH | DAY | YEAR | HOURS | MINUTES | SECONDS
------|-----|------|-------|---------|---------
STRING | INTEGER | INTEGER | INTEGER | INTEGER | INTEGER

4E IS DEFINED

HIERARCHY

CHANGE NUM

HOMOGENEOUS: YES
BASIC ITEMS: YES
ORDERED: NO
NUMBER: FIXED, 2
IDENTIFICATION: NAME

BUGS | ENHANCE
-----|------
INTEGER | INTEGER

LEVEL 4
LATEST RELEASE IS SINGLE MOST RECENTLY PRODUCED RELEASE OF FINAL PRODUCT.
CURRENT RELEASE IS ALL CURRENT RELEASES EXCEPT MOST RECENT ONE.
FIXED IS RELEASE IN WHICH TR IS FIXED.

LEVEL 5
2 IS REDEFINED

6A IS DEFINED

RELEASE INFO

C=YES

1

5A

1

F I X E D

HOMOGENEOUS: NO
BASIC ITEMS: NO
ORDERED: NO
NUMBER: FIXED, 3
IDENTIFICATION: NAME

HIERARCHY

AFFE C TS

M

PROGRAM

1 C

VERSION

1 D

6B IS DEFINED

RECEIVED

DATE-TIME

4 D

STATUS

TABULAR

1-3

DESCRIPTION

STRING

STATUS IS 1 IF NEITHER FIXED OR AFFECTS EXISTS FOR THIS TR.
2 IF ONLY AFFECTS EXISTS.
3 IF BOTH AFFECTS AND FIXED EXISTS.

LEVEL 6 PART 1
APPENDIX D

GLOSSARY OF TERMS

This appendix is a glossary of significant terms defined as they are used throughout the thesis. Within the body of the text, these terms are underlined when first discussed.

Additional structure decision (in top-down data structure design): the decision, at some level of a top-down design, to introduce further data organization among the existing data definitions.

A-end (of an association): one of the two end points of an association; the two ends are not distinguished by direction or any kind of superior/subordinate relationship.

Aggregate: a data structure which groups together a collection of separate data elements into a single table-like structure.

Aggregate data definition: a listing of the kinds of data elements and a description of how they are grouped together to form an aggregate.

Aggregate instance: an acceptable number of instances of the various data elements listed in an aggregate data definition correctly grouped together.

Analysis model: a data structure model consisting of a compilation of or framework for all possible variations among a collection of real world data structures.

Association: a pairing or binary relation between aggregates and basic items.

Association data definition: a specification of the data definitions of the aggregates and basic items to be associated and the details of their interrelation.
Association instance: an appropriate number of aggregate and basic item instances interrelated in the manner described by an association data definition.

Basic item: a primitive, indivisible data element.

B-end (of an association): one of the two end points of an association; the two ends are not distinguished by direction or any kind of superior/subordinate relationship.

Components decision (in top-down data structure design): the decision, at some level of a top-down design, to break up an abstraction into one or more components and to group them together according to some specific data structuring technique.

Data definition: a special declaration to describe the organization, format, and structure of part of a data base.

Dictionary (of data definitions): the collection of explicit data definitions for all the parts of a data base.

Entry (of a file): the part of a data structure selected by a file.

File: a way of selecting or picking particular instances of some specific part of a data structure; the part is the file's entry.

File data definition: a specification of a particular data structure, including one or more entries, and rules for selecting particular entry instances.

File instance: any number of instances of the data structure specified in a file data definition and a particular set of rules for picking entry instances.

Grouping decision (in top-down data structure design): the decision, at some level of a top-down design, to collect or group together certain kinds of information.

Information model: a formalism used to study and model real world information; this formalism may have some aspects of a data structure.
Instance (of a data definition): a piece of information formed according to a specific data definition.

Prototype model: a structure model consisting of one formal or abstract construct which can be used to imitate or investigate real world data structures.

Redefinition: a definition of a formerly described data structure from a new, alternative viewpoint which requires different details or conceptual organization.

Redefinition decision (in top-down data structure design): the decision, at some level of a top-down design, to completely change the structure of some previous level using a redefinition.

Restatement: an expression, in English or a simple picture, of the current view of some earlier level of a top-down design in light of the further refinements made since its definition.

Semantic model: a data structure model which defines a data structure by describing its access functions; these access functions completely characterize the data structure and provide the user's only interface to it.

Structure model: a data structure model which describes the static, unchanging, structural aspects of a data structure independently from any access of it.

Top-down design: a design based on levels making use of abstractions which will be described in a different level; each level is a readable, understandable entity which can be considered in a stand-alone fashion.
REFERENCES


Pfaltz, J.L. 1972. Graph structures. J. ACM 19, 3 (July 1972), 411-422.


Name: William Leonard Honig

Place of birth: Detroit, Michigan

Date of birth: October 8, 1947

Degrees:
- Master of Science in Information Science
  University of Chicago
  Chicago, Illinois
  June, 1971

  Bachelor of Science in Mathematics
  University of Michigan
  Ann Arbor, Michigan
  May, 1969

Employment:
- Member of Technical Staff
  Bell Laboratories
  Naperville, Illinois
  May, 1969 to present

Publication:
Bringing data base technology to the programmer.