December 1971

An Approach to Data Description and Conversion

Diane P. Smith
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-72-20.

An Approach to Data Description and Conversion

Abstract

Currently, the structure of stored data is determined implicitly by the software which accesses and processes it. This data structuring technology has given rise to two outstanding problems in data processing. First, there is the communication of the exact structure of data to users and machines, and secondly, the interchange of the data itself.

This work contributed to overcoming these problems by developing a technique for describing the structure of data explicitly and independently of machines and software. This aim is reflected in the following objectives:

1) To understand data structures by developing a model which not only characterizes current data organizational techniques, but also provides a framework within which new data structures can be defined.

2) To use this model to develop a language which can explicitly describe the organization of data.

3) To use this model to study how data can be converted from one structure to another, with a view towards developing a method for describing data conversions.

This model unifies the diverse area of data structures by including the record, file and storage organizations of data. Furthermore, the model clearly separates at each level the conceptual part, which is the logical structure imposed by a user, from the implementation part, which is the method by which the logical structure is I encoded as a binary representation. This separation leads to n straightforward mapping of a file onto storage. From an analysis of the state-of-the-art in data organization, it is shown that the model can express not only the data structures of current systems, but also certain useful generalizations which might well be produced by future systems.

The model treats records as hierarchies of data items. These hierarchies are expressed by production systems based on a generalized notion of attribute-value pairs. Files are treated as graphs whose nodes are records. The connections between the nodes are expressed using a powerful production system which generates criteria for determining when any two records are to be linked. The structure of storage is generalized as a hierarchy since this structure is common to all storage media. The mapping of files onto storage is expressed in terms of rules for distributing the records of the file within the slots provided by the storage structure.

The language, called Generalized Data Description Language (GDDL) is a realization of the model, and thus possesses all its capabilities . In particular, the language can describe the implementation of any aspect of a file as being dependent on any other aspect. The language is presented in an appendix in the form of a user's manual.

Data conversion is studied in terms of transforming data in one structure to another, where both structures are expressed in the model. This study shows that to fully specify a conversion the relationship between the components of the two structures must be specified. In certain cases, such as the reorganization of a file, this relationship can be very elaborate. A method is developed for specifying such relationships, and a corresponding capability is built into GDDL. Thus, WDL has the ability not only to fully describe data structures, but also to specify data conversion.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-

CIS-72-20.

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

TECHNICAL REPORT

AN APPROACH TO DATA DESCRIPTION AND CONVERSION

by

Diane Pirog Smith

Project Supervisor
Noah S. Prywes

December 1971

Prepared for the
Office of Naval Research
Information Systems
Arlington, Va. 22217

under

Contract N00014-67-A-0216-0007
Project No. 049-272

Moore School Report No. 72-20

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
University of Pennsylvania The Moore School of Electrical Engineering Philadelphia, Pa. 19104	UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

AN APPROACH TO DATA DESCRIPTION AND CONVERSION

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report

5. AUTHOR(S) (First name, middle initial, last name)

Diane Pirog Smith

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1971	328	20

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-67-A-0216-0007 b. PROJECT NO. NR 049-272 c. d.	Moore School Report No. 72-20 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Reproduction in whole or in part is permitted for any purpose of the United States Government.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Office of Naval Research Information Systems Arlington, Virginia 22217

13. ABSTRACT

Currently, the structure of stored data is determined implicitly by the software which accesses and processes it. This data structuring technology has given rise to two outstanding problems in data processing. First, there is the communication of the exact structure of data to users and machines, and secondly, the interchange of the **data itself**.

This work contributes to overcoming these problems by developing a technique for describing the structure of data explicitly and independently of machines and software. This aim is reflected in the following objectives:

1) To understand data structures by developing a model which not only characterizes current data organizational techniques, but also provides a framework within which new data structures can be defined.
2) To use this model to develop a language which can explicitly describe the organization of data.
3) To use this model to study how data can be converted from one structure to another, with a view towards developing a method for describing data conversions.

The model unifies the diverse area of data structures by including the record, file and storage organizations of data. Furthermore, the model clearly separates at each level the conceptual part, which is the logical structure imposed by a user, from the implementation part, which is the method by which the logical structure is encoded as a binary representation. This separation leads to a straightforward

(continued)

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Data Description						
Data Structure						
Record Structure						
File Structure						
Storage Structure						
Data Conversion						
Access Paths						
Attribute						
Value						
Field						
Group						
Data Item						
Association List						

DD FORM 1473

Abstract (continued)

mapping of a file onto storage. From an analysis of the state-of-the-art in data organization, it is shown that the model can express not only the data structures of current systems, but also certain useful generalizations which might well be produced by future systems.

The model treats records as hierarchies of data items. These hierarchies are expressed by production systems based on a generalized notion of attribute-value pairs. Files are treated as graphs whose nodes are records. The connections between the nodes are expressed using a powerful production system which generates criteria for determining when any two records are to be linked. The structure of storage is generalized as a hierarchy since this structure is common to all storage media. The mapping of files onto storage is expressed in terms of rules for distributing the records of the file within the slots provided by the storage structure.

The language, called Generalized Data Description Language (GDDL) is a realization of the model, and thus possesses all its capabilities. In particular, the language can describe the implementation of any aspect of a file as being dependent on any other aspect. The language is presented in an appendix in the form of a user's manual.

Data conversion is studied in terms of transforming data in one structure to another, where both structures are expressed in the model. This study shows that to fully specify a conversion the relationship between the components of the two structures must be specified. In certain cases, such as the reorganization of a file, this relationship can be very elaborate. A method is developed for specifying such relationships, and a corresponding capability is built into GDDL. Thus, GDDL has the ability not only to fully describe data structures, but also to specify data conversion.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my two supervisors: Dr. David K. Hsiao who first introduced me to this area of research and who provided invaluable help and careful criticism, and Dr. Grace Murray Hopper whose conviction of the importance of the topic provided the encouragement I needed and whose vast experience in the area helped me to recognize many of the crucial aspects of the problem. I would also like to thank Dr. Noah S. Prywes and Dr. James Emery for their support and guidance.

The Ford Foundation and the U.S. Army Electronics Command, Avionics Project, supported me at various times during my graduate studies. I am particularly grateful to the Information Systems Branch of the Office of Naval Research for supporting this research under contract N00014-67-A-0216-0007.

INDEX

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF TABLES

BIBLIOGRAPHY

(Bi 1948) Birkhoff, G., Lattice Theory, American Mathematical Society, 1948.

(Ch 1968) Chapin, N., "A Deeper Look at Data," Proceedings 1968, ACM National Conference, 1968, pp. 631-638.

(CO 1971) CODASYL Data Base Task Group, Data Base Task Group Report to the CODASYL Programming Language Committee, April 1971.

(CO 1969) CODASYL Systems Committee Technical Report, A Survey of Generalized Data Base Management Systems, May 1969.

(Co 1970) Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Volume 13, Number 6 June, 1970, pp. 377-387.

(Ga 1970) Galler, B.A. and Perlis, A.J., A View of Programming Languages, Addison-Wesley, 1970.

(Hs 1970) Hsiao, D. and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, pp. 67-73.

(Hs 1971) Hsiao, D., "A Generalized Record Organization," IEEE Transactions on Computers, December 1971.

(IBM 1965) IBM System/360 Operating System, PL/I Language Specifications, File No. S360-29, Form C28-6571-4, 1965.

(La 1968) Lancaster, F.W., Evaluation of the MEDLARS Demand Search Service, U.S. Department of Health, Education and Welfare, Public Health Service, National Library of Medicine, Bethesda, Maryland, January 1968.

(Ma 1971) Manola, Frank, "An Extended Data Management Facility for a General Purpose Time Sharing System," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, 1971.

(Ma 1969) Marden, E., "Statement of Need for a Data Descriptive Language," Statement prepared for USA Standards X3 Ad Hoc Committee, 1969.

(Me 1967) Mealy, G., "Another Look at Data," FJCC, 1967, pp. 525-534.

(Ra 1971) Ramirez, J., and Solow, H., "The Design and Implementation of the DDL Processor," The Moore School of Electrical Engineering, University of Pennsylvania, work in progress.

(RCA, 1969) RCA Information Systems, COBOL Reference Manual, 70-00-607, May 1969.

(RCA, 1970) RCA Time Sharing Operating System, Data Management System Reference Manual, DJ-001-2-00, June 1970.

(Sa 1969) Sammet, Jean E., Programming Languages: History and Fundamentals, Prentice-Hall, 1969.

(St 1967) Standish, T.A., "A Data Definition Facility for Programming Languages," Carnegie Institute of Technology, 1967.

(SSDL 1970) Storage Structure Definition Language Task Group, "Storage Structure Definition Language, SSDL," Record of the 1970 ACM SICFIDET Workshop on Data Description and Access, Rice University, Houston, 1971.

(US 1968) U.S. Navy Programming Languages Group, Fundamentals of COBOL, NAVSO P-3063, 1968.

CHAPTER 1 INTRODUCTION

1.1 Background and Objectives

Computer technology is a field which has experienced a rapid and uneven evolution. This evolution has seen computer users develop techniques and conventions appropriate only to their own needs and data processing environments. This has led to the inability of different user groups to communicate information about, and to exchange algorithms and data effectively. The problem of user and machine dependent algorithms has received considerable attention, resulting in the development of widely accepted and largely machine independent programming languages such as ALGOL. However, the severity of the problems of user and machine dependent data organization has only been realized comparatively recently*, and as yet little has been done to alleviate this situation.

Traditionally data is organized either by developing special software or by specifying its structure in existing programming languages, operating systems or data management systems. In either case, the exact data organization can only be understood by analyzing and interpreting several complex and interacting programs written in a variety of languages. For example, to understand the data structures produced

---

* "It has been estimated that the lack of an adequate data description language is costing the Department of Defense alone millions of dollars annually because of the inability to exchange data effectively." (Ma 1969, pg. 1)

by a particular COBOL program, it is necessary to analyze and interpret the following programs:

(i) the COBOL program itself,

(ii) the COBOL compiler, and

(iii) the data management system of the machine being used.

This effort is necessary because the factors which determine the organization of data are implicit in the programs and software used to process and structure the data. Consequently, such practices in data organization have hampered not only the communication of data structures but also the interchange of the data itself. When data is to be interchanged, it is necessary to know first whether the existing organization is compatible with the new software which is to use it, and secondly, how the organization can be converted to make it compatible when this is not the case. The implicit nature of data organization can make this an onerous task.

A solution to these problems of communication and data interchange is to make the organization of data explicit and its understanding independent of machines and software systems. This can be achieved by developing a language for explicitly specifying data structures which is separate from the languages used to process that data. To understand a data structure, it is then only necessary to interpret a specification which is expressly intended to communicate data structure information, rather than to interpret a program one of whose side effects is the structuring of data.

Such a data description language (ddl) would have many applications. One important application is to provide a means of communicating data structures among users. For example, using a ddl a creator of a data base can describe precisely to an applications programmer the exact structure of the data that the programmer wants to use. Just as ALGOL is now used to communicate algorithms so can a ddl be used to communicate data structures.

Not only can a ddl be used to communicate with users, but by constructing a ddl interpreter, the ddl can be used to communicate with machines. Using such an interpreter, a computer could use the information contained in any file when it is provided with a ddl description for that file. Users would then be free to structure their data in whatever manner they deem appropriate, without being constrained by the data structure specification facilities available in operating systems and programming languages. Thus, a ddl could be used in establishing automatically the structure of data bases. A data base creator would provide a ddl description and his data to the interpreter which would structure the data according to the description.

Furthermore, we could apply a ddl to the problem of mechanizing the conversion of data from a current structure to a new structure. It would only be necessary to input to a converter the data, a ddl description of its current structure, a ddl description of its new structure and a ddl description of the relationship between elements in one structure and the other. By interpreting these descriptions the converter could output the data in its new structure. Thus, the

user is released from writing special conversion programs. In this way files could be interfaced across programming language, operating system, data management system and hardware barriers.

A further application is in the design and operation of data and data base management systems. For example, a ddl can be used to create new data structures which can then be tested for effective storage utilization and other efficiency considerations.

At this point we should make clear what we mean by the term "data structure". We use the term to refer to the structure of data as it is to appear on a storage medium, including both the conceptual organization imposed by the user and the implementation of this conceptual organization. Some research groups, particularly those in programming languages (St 1967, Ga 1970), often use data structure to refer to not only the structure of data (as we use the term) but also the access method by which this data is used. To these groups a pushdown, for example, is a data structure, whereas we would say that a pushdown is a data structure together with an access method which controls storage and retrieval on a last in - first out basis. An access method is a program which is designed to store and retrieve data from a data structure. It follows from our discussion above that we need to separate out data structures from the programs which use them, so we can describe the data structures independently and explicitly. Furthermore, any appropriate access method can be designed once the data structure has been specified.

With this background in mind, we state three objectives for this dissertation:

1) To understand data structures by developing a model which not only characterizes current data organizational techniques, but also provides a framework within which new data structures can be defined.

2) To use this model to develop a language which can explicitly describe the organization of data.

3) To use this model to study how data can be converted from one structure to another, with a view towards developing a method for describing such conversions.

It is anticipated that data description languages will contribute as much as programming languages towards the evolution of information processing. Just as the current state of programming languages is the accumulation of many efforts, it is expected that much research and development will be needed to fully understand the power and applicability of data description languages. The development of the ddl in this dissertation is perhaps analogous to the development of the first programming language. Different programming languages usually have different models of algorithms on which they are based. For example, ALGOL is based on recursive procedures with arithmetic operations, whereas LISP is based on the lambda-calculus and string manipulations. Similarly, we provide our own model of data organization on which our data description language is based.

There are other studies in progress which relate to the design of a ddl, specifically the studies being made by the CODASYL Storage Structure Description Language Task Group (SSDL 1970). However, this group so far has mainly addressed itself to techniques for mapping records onto storage, which is just a subset of the problem we have tackled here. The language given here is the first one to be completely developed and specified. In addition, we are the first to study and propose a general solution for the problem of using data descriptions for converting data from one structure to another.

1.2 The Development of the Model, the Design of the Language, and the Study of Conversion

We will now discuss the development of the model and its use in the design of the ddl (called GDDL for Generalized Data Description Language) which is presented in this report.

The development of data description from its first primitive forms in machine languages to its current forms in data management systems has been based on ad hoc changes triggered by user needs and new technology. This has led to a wide variety of methods for describing data, without any general concept or comprehensive model. For example, COBOL (US 1968) is based on highly developed record concepts, whereas L6 (Sa 1969) is based on certain aspects of list structures, and in operating system design, systems programmers have built up a body of expertise on storage structures and file implementation techniques. However, the common concepts underlying these and other aspects of data structures have not been extracted and formulated into a comprehensive model.

Therefore, a thorough study of the data description elements in software systems and programming languages was undertaken, with a view towards extracting those common elements to include in a comprehensive model of data structures.

This model of data structures is divided into three largely independent levels, namely, the record, file and storage levels, and each level is further subdivided into a conceptual part and implementation part. The conceptual part is the logical structure which is imposed on the data. The implementation part is the way in which this structure is to be represented or encoded. The components of this subdivision of data structures are illustrated in Figure 1-1.

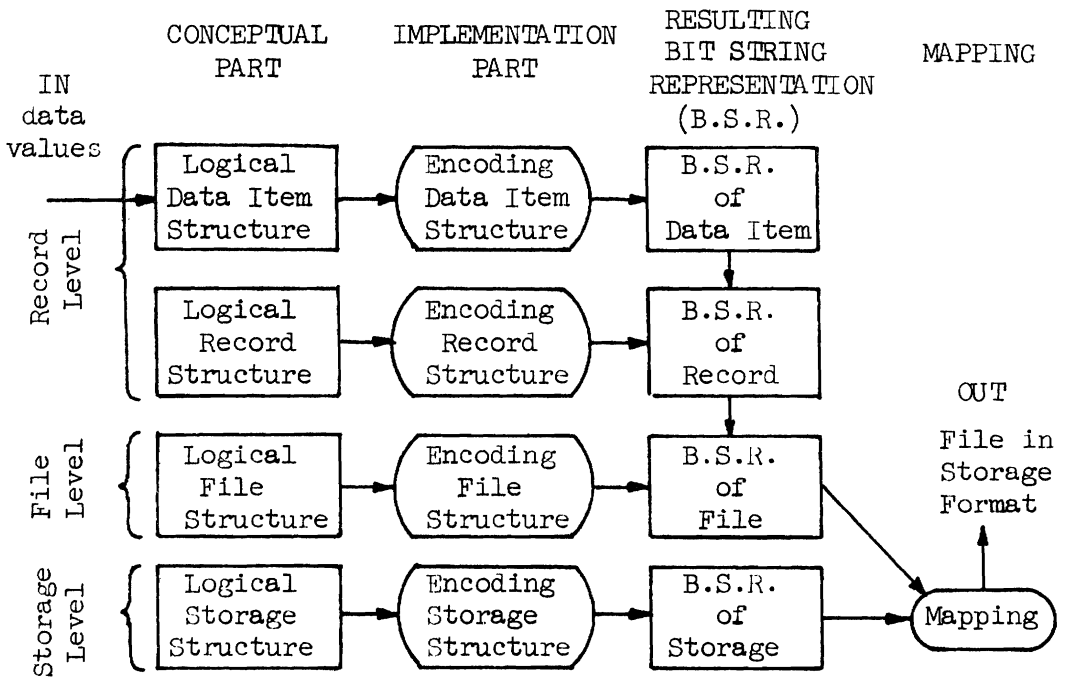


Figure 1-1. The Components of a Data Structure and their Interrelationships

These subdivisions provide a valuable vantage point for understanding data structures. Let us look first at the implications of the division into conceptual and implementation parts.

The nature of the conceptual part is quite distinct from the implementation part, even though most systems do not make this distinction. The conceptual part is the machine-independent structure which is imposed on the data by the user. He conceives of the data as being organized in this fashion, and this is the form in which his programs expect to find the data. The implementation part, which is machine-dependent, is the way in which the logical structure is encoded as a bit string representation which can be stored on a storage medium. In our model we will see that specifications which relate to the conceptual part have the nature of production systems, whereas, specifications which relate to the implementation part have the nature of certain characteristics of character strings like length or character code.

In addition, this subdivision yields a valuable insight which has not been noted in other work. This insight is based on the observation that if a person intends to organize certain entities into a structure, he may want that organization to depend on any property of those entities which are available to him. In particular, if a person wants to organize records into a file, he may specify this organization in terms of any available properties of those records. These properties can include the values of data items in records, the logical structure of the records and the implementation of the record structure. Thus we can

see that to describe file organization we have to provide more than the capability of just specifying abstract graphical structures.

Now we look at the implications of dividing the model into record, file and storage levels.

The concept of a record is common to all data storage and retrieval systems, yet it is usually overlooked in theoretical studies of data structures. The structure of records is an important consideration in that it is the basic organization of data items which is treated as an entity for storage and retrieval. Thus far a hierarchic organization for records has proven adequate, as it provides a structure which is relatively easy to encode and decode without the need for extended scanning operations. In this work, therefore, we only allow hierarchic structures at the record level. In our model this hierarchic organization is generalized in that it allows for levels of the hierarchy to occur optionally or to repeat a number of times. This conceptual structure of records has not been modelled explicitly before, although it is essentially the logical organization of records which is implicit in COBOL. COBOL, however, is quite restrictive on the ways in which the implementation of records may be specified. In this work we allow each implementation characteristic to be specified either directly or dependent on other characteristics.

Records are the elements which are organized into files. There is great flexibility in distributing the overall organization of a set of data items between the record and file levels. On one hand, we can specify a record to consist of a single data item, and, in effect,

specify the overall organization of the data at the file level. In fact we can specify hierarchies at the file level and thus all the conceptual structure for records can in principle be moved to the file level. However, while the conceptual structure of the data might remain the same, the use of the data for storage and retrieval has been changed. On the other hand, we can specify a record to be a complex hierarchic structure and possibly make the file structure simple. The distribution of structure between the file and record levels depends on the intended use of the data. Therefore, by distinguishing record structure from file structure we are able to include these aspects of data structures in our model.

Our concept of a file structure is more general than others because, as previously mentioned, we allow the specification of graphical structures which depend on data and record properties. This requires a more elaborate specification method than the usual methods based on pure graph-theory.

The specification of the structure and encoding of records, and the specification of how these records are structured and implemented as a file determine a bit string representation of the file. This is the bit string which is actually mapped onto a storage structure.

Our division of storage structure into conceptual and implementation parts is the key to both simplifying the mapping of the bit string representation of a file onto a storage structure, and also simplifying the specification of storage structures by extracting the structure

common to storage media independent of physical considerations. The conceptual structure of storage is based on generalized hierarchies which are common to all storage media. The implementation of these hierarchies is based on encoding characteristics which are also independent of the storage media. To bind a storage structure to a particular medium, we have only to relate the levels of the hierarchy to the actual physical levels of a storage medium.

It is over such a storage structure that the bit string representation of a file is distributed. A result of our subdivision of data structures has been to make the actual mapping of data onto a storage medium comparatively straightforward. It is only necessary to deconcatenate the bit string representation of the file at appropriate points, and insert these component strings without disturbing their order into the slots already provided by the storage structure.

These are the insights and advantages which are obtained by subdividing our model in the above way. From the study of data description elements in software systems and programming languages we can ensure that we at least included the data description capabilities of every current system that was considered. As each of the classes of software in the study includes the most sophisticated representative of that class, it is likely that we have in fact included the capabilities of all current systems. From this model the requirements for a data description language are immediately apparent. This allows GDDL itself to be very closely related to the model.

When the data description capability of the language had been designed, the problem of using descriptions to convert data from one structure to another was studied. Using ddl's for data conversion is one application that has been widely suggested, but never actually investigated. With our model of data structure, we could study the conversion process itself. In this study it will be shown that additional information is required to completely describe a conversion. This additional information specifies a relationship, which can be quite elaborate, between names in one description and names in the other. To model this relationship the concept of an association list was developed. GDDL capabilities for describing data conversion relationships are incorporated directly from the association list concept.

1.3 Organization of the Report

The GDDL language itself is presented in Appendix A in the form of a self-contained reference manual. The body of this report therefore is concerned with presenting the model and its relationship to the language. It also shows that GDDL can describe any data organization that can be obtained with current systems. Further, because the model allows generalizations of current data description capabilities, GDDL can describe data organizations that are beyond these present capabilities but might well be incorporated into future systems. The generality of GDDL relative to current systems is discussed in terms of the model.

Chapter 2 presents the study of the development of data description in programming languages and software systems. The table at the end of this study (Table 2-1) provides the basis for showing that the models and thence GDDL include all current data structure capabilities. This study is quite long and the details are not essential for understanding the remaining chapters. The reader is therefore advised to skip to Chapter 3 should the detail become too oppressive.

Chapters 3, 4 and 5 develop the record, file and storage levels of the model respectively. Each chapter shows the relationship between the model and the GDDL language at that level. The material in these chapters provides an excellent way of visualizing the structure of GDDL and its description capabilities.

Chapter 6 discusses the ways of using data descriptions to convert data from one structure to another. The concept of an association list is introduced and it is shown how an association list can be used to complete the specification of data conversion.

Chapter 7 summarizes the contributions of this report and suggests directions for future research.

Appendix B contains examples of GDDL descriptions of some real-world files and of data conversion from one structure to another. These examples are chosen to further demonstrate the ability of GDDL to describe current data organizations.

Appendix C contains a proof that GDDL can indeed describe all the COBOL record features. COBOL is the prototype for the most advanced record level data representations. It is shown that each COBOL record

description clause can be expressed in GDDL.

CHAPTER 2 EXISTING DATA STRUCTURES AND DATA DESCRIPTION LANGUAGES

2.1 Introduction

The object of this chapter is to provide an analysis of data structures in contemporary computer software with a view towards obtaining a comprehensive summary of data structure characteristics. This summary provides the basis for demonstrating in later chapters that the CDDL is complete.

The software systems covered by this analysis are:

(i) machine languages,

(ii) early operating systems,

(iii) assembly languages,

(iv) early higher-level programming languages,

(v) current operating systems,

(vi) current higher-level programming languages,

(vii) data base management systems, and

(viii) the CODASYL Data Description Language.

The characteristics of each of these systems are analyzed in a separate section of this chapter. The final section combines the results of these analyses into a table.

2.2 Data Structures in Machine Languages

In machine languages, there are four ways that data structure characteristics are specified:

1) hardware specifications for conventions such as the code for representing characters, the base for representing numbers, and the length of the smallest addressable unit of storage. These conventions are fixed for a given computer but may vary from machine to machine. To use a particular machine, a system programmer has to know these conventions. Thus, descriptions in the form of specifications in manuals are usually provided.

2) machine language instructions that specify the data type (e.g., character or number), the scale of numbers (e.g., fixed point or floating point), and the precision of numbers (e.g., single or double). These descriptive elements are implicit in data manipulation instructions rather than explicit as declarations. They are illustrated by the following examples.

a) To specify that a character string is to be placed in the accumulator of the computer, the machine language instruction CAL (Clear and Add Logical Word) would be used instead of the instruction CLA for placing a number in the accumulator.

b) To specify that a floating point number is to be added to the accumulator, the instruction FAD (Floating Add) would be used instead of the fixed point instruction ADD.

c) To specify double precision for addition, the instruction DFAD (Double Precision Floating Add) would be used instead of the single precision instruction ADD.

3) machine language instructions that specify locations of data items. These descriptive elements are also implicit in data manipula-

tion instructions rather than explicit as declarations. For example, the STO (Store) instruction both declares that a particular location is to be used for storage and specifies that a data item is to be stored in that location.

4) machine language instructions that specify which devices are to be used for input and output, and how data would be organized on the device medium. These descriptive elements are also implicit in data manipulation instructions rather than explicit as declarations. They are illustrated by the following examples.

a) To specify that a particular I/O device is to be used for output, the machine language instruction WRS (Write Select) is used to prepare the appropriate channel.

b) To specify that a particular block of data items is to be copied onto an output medium, the instruction RCH (Reset and Load Channel) is used to send to the channel a channel command word which gives the size of the block of data to be copied and its location.

c) To specify that the last block of data has been reached on a magnetic tape, the instruction WEF (Write End-of-File) is used to write an end-of-file gap followed by a tape mark on the tape.

The characteristics of data structures* provided by machine languages can be grouped into two categories. One includes the characteristics of individual data items, and the other the characteristics

---

* At the end of each section of this chapter a list of the characteristics of the system under discussion will be presented. Whenever a new characteristic (not appearing in previous sections) is introduced, it will be underlined.

of storage media.

1. The characteristics of individual data items consist of:
   - (i) the hardware provided character code,
   - (ii) length,
   - (iii) data type:
     - a) character string,
     - b) numbers:
       - 1) binary base,
       - 2) Sign - radix or diminished radix complement (depending on the hardware),
       - 3) fixed or floating-point scale.
2. The characteristics of storage media consist of:
   - (i) block size,
   - (ii) end-of-file labels, and
   - (iii) device assignment.

We note that machine instructions are seldom used or made available to describe explicitly the structuring of sets of data items. Such structures are created and maintained by machine language programs.

2.3 Data Structures in Early Operating Systems

With the development of Operating Systems (OS's), more complex data structures on storage devices were provided directly to the programmer. They are described by statements of the OS job control language (JCL). Previously, these file and storage structures had to be implemented as part of user-written machine language programs.

Examples of such statements are the $FILE and $LABEL statements provided by the IBM 7040 JCL. These are illustrated in Figure 2-1.

The $FILE statement is used to describe the characteristics of the file structure and the positioning of the records on magnetic tape, the structure of the tape's physical blocks and the tape unit.

1. The file structure and implementation characteristics consist of:
   (i) ordering the records in their input sequence, and
   (ii) implementing this structure by sequential storage.
2. The record positioning characteristic is the record to tape block ratio; that is, the number of records per tape block.
3. The storage structure and implementation characteristics are:
   (i) tape naming,
   (ii) tape block size,
   (iii) labels:
      a) header and trailer labels for tape reels and files,
      b) count fields for tape blocks,
   (iv) fixed ordering of tape blocks and labels on the tape,
   (v) fixed occurrence of all blocks and labels specified,
   (vi) repetition of reels - given as number of reels.
4. The device characteristic is read/write density.

The remaining parameters of the statement are used to describe buffers and actual processing.

The $LABEL statement is used to describe the information in a label. Labels are used to implement storage structures.

```
1       8              16
-------------------------------------------------------------------
$FILE   deck name 'file name',[primary unit],[secondary unit]
```

```
1                16
-------------------------------------------------------------------
$ETC             BLOCK=xxxx [,{SINGLE}][,{REEL}] [,{HIGH}]
                            {DOUBLE}  {REELS}  {LOW}
```

```
1                16
-------------------------------------------------------------------
$ETC             [MIXED] [,{SEQ}  ][,{CKSM}  ] [,{PRINT}]
                           {NOSEQ}  {NOCKSM}    {PUNCH}
                                                {HOLD}
                                                {SCRTCH}
```

```
1                16
-------------------------------------------------------------------
$ETC             [{ADDLBL=exname}] [,LRL=xxxx] [,RCT=xxxx]
                  {NSLBL=exname}
                                               [,EOR=exname]
```

```
1                16
-------------------------------------------------------------------
$ETC             [ERR=exname] [,{TYPE1}] [,EOF=exname]
                              {TYPE2}
                              {TYPE3}
```

Figure 2-1 a) The IBM 7040 $FILE Statement

```
1                16
-------------------------------------------------------------------
$LABEL           'filename', [file   ] ,[reel    ] ,[{date} ] ,
                             [serial ]  [sequence]  [{days}]
                             [number ]  [number  ]
                                                    identification
```

Figure 2-1 b) The IBM 7040 $LABEL Statement

Figure 2-1. IBM 7040 Data Description Statements

2.4 Data Structures in Assembly Languages

Assembly languages were primarily designed to enhance data handling and to a lesser degree, to provide mnemonic machine instructions. The data-oriented pseudo-instructions provided by assembly languages significantly increase the variety of data structures made directly available to the user. Thus, many complex data structures that had previously been created and maintained by user programs, can now be declared explicitly.

In Assembly Languages, elements and statements which deal with data structures are typified as follows:

1) Symbolic names assigned to data items. These names may be used to access the data items directly without referring to the address of the data items. For example, in the IBM 7040 Macro-Assembly Language MAP, the statement DDATA DEC 13 results in the name DDATA being assigned to the location in which a decimal number 13 is stored.

2) Pseudo-instructions that declare data types. For example, in IBM 7040 MAP, data items may be declared to be octal, OCT; decimal, DEC; binary coded information, BCI; and variable field data, VFD. This is illustrated by the following examples:

a) To specify that a data item named DDATA is to be interpreted as the decimal integer 13, the following MAP statement is used:

```
DDATA DEC 13
```

b) To specify that a data item named ENTRY is to contain the character C in the first 6 bits of the data item, the following MAP statement is used:

ENTRY VFD H6/C

3) Pseudo-instructions that describe the structure of data items. For example, in IBM 7040 MAP, to specify that a block of 6 consecutive storage locations are to be reserved for storing data items, the following statement is used:

BSS 6

4) Pseudo-instructions that describe input/output characteristics of particular media. For example, in IBM 7040 MAP such statements are of the form:

name FILE option, ..., option

LABEL option, ..., option

where the options for the FILE statement and LABEL statement are the same as the options for the IBM 7040 Job Control Language $FILE and $LABEL described in the previous section.

Thus, the following characteristics of individual data items, sets of data items and storage media are made accessible to programmers in Assembly Language.

1. The characteristics of individual data items consist of:

   (i) symbolic naming,

   (ii) the hardware provided character code,

   (iii) length,

   (iv) data type:

      a) character string,

b) numbers:

1) binary, decimal or octal base,

2) character sign for decimal numbers and radix and diminished radix complement for binary numbers,

3) fixed or floating point scale,

(v) data items identified by position.

2. The characteristics of sets of data items consist of:

(i) fixed order,

(ii) fixed occurrence, and

(iii) sets of data items identified by their position

3. Assembly languages depend on their underlying operating system for storage structure.

2.5 Data Structures in Early Higher-Level Programming Languages

In developing higher-level languages such as FORTRAN and COBOL, appropriate data structures were provided. For example, FORTRAN, which was designed for scientific computing, provides array accessing for handling homogeneous data (i.e., data of the same type).

The data description statements of ANSI FORTRAN have four forms:

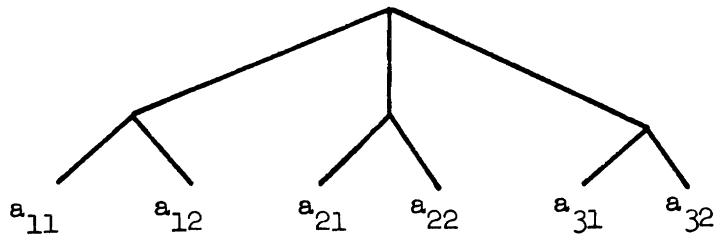
1) Declaration statements that describe the structure of individual data items. In FORTRAN, characteristics such as scale and precision are treated as additional data types. For example, in FORTRAN IV, the following "type" declarations are provided:

INTEGER	DOUBLE PRECISION
REAL	LOGICAL
COMPLEX	EXTERNAL

where LOGICAL data items are the values T (or TRUE) and F (or FALSE), and EXTERNAL data items are data items which are defined externally to the FORTRAN program. To specify the type of a data item, the name of the item is listed after the type in a declaration statement, e.g.,

INTEGER CVAL, A, B

2) The declaration statement which describes the structure of sets of data items (groups). In ANSI FORTRAN, individual data items can be grouped together in hierarchic structures which are interpreted by the processor as arrays. For example, the tree illustrated below can be interpreted as a 2 x 3 array:



That is, the pairs of data items $< a_{11},a_{21} >$, $< a_{21},a_{22} >$ and $< a_{31},a_{32} >$ are interpreted as rows. Arrays are limited to a maximum of three dimensions. The DIMENSION statement is used to describe such groupings. The statement has the following format:

DIMENSION array name (n_1,n_2), ..., array name (n_1,n_2,n_3)

where: array name is the name used to refer to the array, and

n_1, n_2, n_3 are the number of elements in each of the dimensions of the array, allowed in ANSI FORTRAN.

For example, the statement:

DIMENSION A(2,3) describes a 2 x 3 array called A.

Data items in the vectors are accessed by array indexing.

3) the FORMAT statement which describes input and output data structures. The statement is used to describe data type and length for each data item in a record to be input or output. For example, in ANSI FORTRAN, the statement has the following format:

FORMAT (data item specification, ..., data item specification)

where a data item specification consists of two parts: a data type and a data length part. These types are:

F real with no exponent

E real with exponent

D real with double precision exponent

I integer

L logical (character string T or F)

A character string

H hollerith (character string used for output only)

Length is given as number of characters per data item. For example, A6 describes a data item which is a string of 6 characters. For real data items, in addition to length, the number of digits to the right of the decimal point is specified. For example, F8.2 describes a data item which is a real number with a maximum length of 8 characters

and which has 2 digits following the decimal point.

4) Input/Output statements that describe the order of the data items to be input or output, and the device to be used. The statements have the following format:

$$\left\{\begin{matrix}\text{READ}\\ \text{WRITE}\end{matrix}\right\} \text{ (device number, format statement number)}$$
$$\text{data name, ..., data name}$$

where: device number refers to a specific device,

format statement number refers to the format statement describing the data items being input or output,

data name refers to the data item or group (array values) being input or output.

Thus, the following characteristics of data structures are made accessible to programmers by the data description statements of FORTRAN:

1. The characteristics of individual data items consist of:
   - (i) symbolic naming,
   - (ii) the hardware provided character code,
   - (iii) fixed lengths as specified by the user,
   - (iv) data type:
     - a) character string,
     - b) number:
       1) binary or decimal base,
       2) radix or diminished radix complement depending on hardware for binary numbers, character sign or no sign for decimal numbers,

3) fixed or floating point scale,

(v) data items identified by their position.

2. The characteristics of records consist of:

(i) array accessing (balanced trees),

(ii) fixed ordering,

(iii) fixed occurrences,

(iv) groups of data items identified by their position.

3. FORTRAN depends on its underlying Operating System for its storage structure.

Because the COBOL language was designed for handling large quantities of data, more importance was given to the data description statements of the language than in FORTRAN. These statements are written in separate sections of a COBOL program. The Data Division is the section for describing the data items, records, files, working storage and program constants. Another section, called the Environment Division, is for describing the storage media. In it, information concerning file selection is given, and the equipment configuration (tape station, printer, etc.) is described.

1) In COBOL's Data Division there is one statement for describing the organization of data items in records and one statement for describing the organization of records into files;

a) Each data item or group of data items that is to appear in a record is described by a statement of the form illustrated in Figure 2-2. This statement is used to describe:

i) the level at which the data item or group of data items is to occur in the hierarchic record,

ii) the data type (e.g., character string = DISPLAY, numeric string = COMP),

iii) the length of the data item,

iv) the number of times the data item or group of data items is to occur in each record,

v) the alignment of the data item in respect to word boundaries and to fixed length strings of character positions.

8	12
level-number	{*data-name-1* / FILLER} [; REDEFINES *data-name-2*] [; USAGE IS {COMPUTATIONAL / COMP / DISPLAY}]
	[; {SYNCHRONIZED / SYNC}] [{LEFT / RIGHT}] [; {PIC / PICTURE} *character-string*]
	[; {JUSTIFIED / JUST} RIGHT] [; VALUE IS *literal*] [; BLANK WHEN ZERO].

Figure 2-2 The ANSI COBOL Statement For Describing a Data Item or a Group in a COBOL Record (US 1968)

b) The organization of COBOL records in a COBOL file is described by a statement of the form illustrated in Figure 2-3. This statement is used to describe

i) the size of storage blocks,

ii) the size of the records stored in the blocks,

iii) any labels to appear on the storage tape,

iv) the names of records appearing in the file.

```
| 8  | 12
|----|-----------------------------------------------------------------------
| FD | file-name [ ; BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS}     ]
|    |                                                      {CHARACTERS}
|    | [; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]
|    | ; LABEL { RECORDS ARE } { STANDARD                                  }
|    |         { RECORD IS   } { OMITTED                                   }
|    |                         { data-name-1 [, data-name-2]. . .          }
|    | [ ; VALUE OF data-name-3 IS {data-name-4} [ , data-name-5
|    |                             {literal-1  }
|    |       IS {data-name-6} ]. . . ]
|    |          {literal-2  }
|    | [ ; DATA { RECORD IS  } data-name-5 [, data-name-6]. . . ].
|    |          { RECORDS ARE}
```

Figure 2-3 The ANSI COBOL Statement for Describing a COBOL File (US 1968)

2) In COBOL's Environment Division, there is one section that is used to describe input and output conventions. In it, equipment assignments and certain physical characteristics of each file to be used by the program are described by a statement of the form illustrated in Figure 2-4. This statement is used to describe the device on which the file is stored.

```
FILE-CONTROL
FILE-CONTROL. { SELECT [OPTIONAL] file-name
ASSIGN TO [integer-1] implementor-name-1 [,implementor-name-2] . . .
[FOR MULTIPLE REEL] [ ,RESERVE { integer-2 } ALTERNATE [ { AREA  } ] ] . } . . .
                               { NO      }               { AREAS }
```

Figure 2-4. The ANSI COBOL Statement for Describing the Storage Convention of a COBOL File (US 1968)

Thus, the data structures that are made accessible to programmers by COBOL can be characterized in the following way.

1. The characteristics of individual data items consist of:
   - (i) symbolic naming,
   - (ii) the hardware provided character code,
   - (iii) fixed lengths as specified by the user,
   - (iv) data types:
     - a) character string,
     - b) number:

1) binary or decimal base,

2) sign - radix or diminished radix complement (depending on the hardware) for binary numbers, and character sign or no sign for decimal numbers,

3) fixed or floating point scale,

(v) value alignment (justification) with blank or zero padding,

(vi) value string alignment (synchronization) with respect to computer words with blank or zero padding,

(vii) data items identified by their position.

2. The characteristics of records consist of:

(i) hierarchic structure,

(ii) fixed order,

(iii) fixed occurrences,

(iv) fixed repetition ordered as input,

(v) groups of data items identified by their position.

3. COBOL depends on its underlying Operating System for its storage structures.

2.6 Data Structures in Third-Generation Operating Systems

In their current stage of development, Operating Systems (OS's) are providing more file and storage structure options than early OS's. The creation and maintenance of these structures are treated as a set of services separate from those involved in scheduling programs. The part of an OS which supports these services is referred to as the

data management system (DMS) of the operating system. Among these services are the moving of data between storage devices and main memory, and the accessing of data in DMS maintained structures. Additional JCL statements, known as DMS statements, are provided to evoke DMS services.

In general, DMS's provide their users with a number of file and storage structures. To store data in such structures, the user proceeds as follows:

(i) he names the particular structure in a DMS statement,

(ii) he lists the parameters which select those options provided by the DMS (if any), and

(iii) he enters his data.

The data management service so evoked moves the data from the input device to the appropriate storage devices and stores it in the described structures.

For example, the DMS II of the RCA SPECTRA 70/46 TSOS (RCA 1971) provides its users with five structures and related input/output conventions. Collectively, these structures and conventions are called access methods. They are:

1) PAM (Primitive Access Method). This method provides only a particular record format (fixed in length) and storage on either direct-access devices or on single reel, standard blocked tape. PAM creates and accesses files only in random order. The user must himself handle the blocking and deblocking of records.

2) SAM (Sequential Access Method). This method provides either fixed length, variable length or undefined record formats (where records with undefined formats are stored one to a block). SAM creates and accesses files in sequential order only. It performs all blocking, deblocking and buffering for the user.

3) ISAM (Index Sequential Access Method). It provides either fixed or variable length record formats and storage on direct-access devices only. Records are maintained by means of a directory whose entries point to the records to reflect the correct sequence. In other words, records may not be in sequential order physically. The field whose values determine the sequence is called the key. Thus, ISAM can access files in a sequential or non-sequential order. In terms of storage structure, an ISAM file is made up of data blocks (2048 bytes) and directory blocks. Data blocks contain the user's records which are ordered initially according to the values of the key field. Directory blocks contain pointers to data blocks. ISAM performs all blocking, deblocking and buffering for the user.

4) BTAM (Basic Tape Access Method). This method provides either fixed length or undefined record formats (where records are stored one per block) and storage on tape only. BTAM is used to provide efficient accessing of tape blocks.

5) EAM (Evanescent Access Method). It provides fixed length record formats and storage on direct-access devices only. EAM creates and accesses temporary files only in a random order. Because they are temporary, EAM files have no labels and require no cataloguing or

security checks.

Data structures in these five access methods are similar in several respects. In fact, only three structures are provided for records:

1) Fixed length - in which each record contains exactly the same number of bytes. Standard format is known to all DMS access methods.

2) Variable length - in which each record may contain a different number of bytes. In each variable length record, the first two bytes of the record contain the characters "ll", and the second two bytes contain the length of the record.

3) Undefined - in which records are identical in length to the input/output buffers defined for the access method.

There are three ways of organizing records into files:

1) random organization,

2) sequential, and

3) indexed sequential.

For storage, records may be blocked and unblocked automatically, devices may be tape or direct-access, and blocks may be standard (2064 bytes) or nonstandard ($\leq$ 4096 bytes). Control codes such as tapemarks, count fields, etc. are handled automatically and may not be specified by the user.

Thus, the following characteristics of file and storage structures are made accessible to programmers by the DMS data description statements.

1. The characteristics for organizing records into files and implementing the structure consist of:
   (i) structuring records by input sequence,
   (ii) structuring records by value (key),
   (iii) implementing structures by
      a) sequential positioning, and
      b) by pointers:
         1) stored in tables or embedded in records,
         2) given as absolute address or relative to some origin.
2. The characteristics for positioning records in device blocks consist of:
   (i) the record-to-block ratio, and
   (ii) the distribution of records such that records either are maintained whole or are split between blocks.
3. The characteristics for organizing storage blocks and implementing this structure consist of:
   (i) block naming,
   (ii) formatting for the following supported devices: magnetic tape, magnetic disk, cards, and printer,
   (iii) block length specification for supported devices,
   (iv) labels for supported devices,
   (v) fixed order of device formats,
   (vi) fixed occurrences of device formats,
   (vii) repetition of formats for tape reels, disk levels,

cards and printer pages.

2.7 Data Structures in Current Versions of Higher-Level Programming Languages

Current higher-level programming languages have been developed to take advantage of the data management services provided by operating systems and to satisfy user requirements for more complex working structures.

For example, RCA SPECTRA 70/46 ANSI COBOL (RCA 1969) has statements to evoke SAM and ISAM and their related data structures.

The COBOL Data Division has been enhanced: new internal formats have been added, repeating groups can be ordered, and repetition numbers can vary for different record occurrences. The clauses used to specify these options are illustrated in Figure 2-5.

```
[ USAGE IS { DISPLAY
           { COMPUTATIONAL
           { COMPUTATIONAL-1
           { COMPUTATIONAL-2
           { COMPUTATIONAL-3
           { INDEX                 } ]
```

Figure 2-5, a. The COBOL Statement for Declaring Data Types

```
[ OCCURS { [integer-1 TO] integer-2 TIMES
         { [DEPENDING ON data-name-1]
         { [ { ASCENDING  } KEY IS data-name-2
         {   { DESCENDING }       [, data-name-3] ... ]
         { [INDEXED BY index-name-1 [, index-name-2] ... ] } ]
```

Figure 2-5, b. The COBOL Statement for Specifying Repetition

Figure 2-5. Enhanced COBOL Description Statements

PL/I is an example of a higher-level programming language that was designed to incorporate a larger number of record structures than other languages available at the time of its conception. It provided array accessing, hierarchic structuring and string processing for data items and groups of data items.

PL/I provides a rich set of characteristics for structuring and implementing data items (IBM 1965):

(i) symbolic naming,

(ii) the hardware provided character code,

(iii) fixed and varying lengths as specified by the user,

(iv) data types:

a) character string,

b) number:

1) binary or decimal base,

2) sign - radix or diminished radix complement (depending on the hardware) for binary numbers, and character sign or no sign for decimal numbers,

3) fixed or floating point scale,

(v) value alignment with zero or blank pad characters,

(vi) data items identified by position.

These data descriptive elements are combined in declaration statements of the form:

i) DECLARE data item name $\left\{ \begin{matrix} \{\text{CHARACTER}\} \\ \{\text{BIT}\} \\ \text{PICTURE} \end{matrix} \right.$ $\left. \begin{matrix} (n)\ [\text{VARYING}] \\ \text{picture string} \end{matrix} \right\}$

or

ii) DECLARE data item name $\left\{\begin{matrix}\text{FIXED}\\\text{FLOAT}\end{matrix}\right\}$ (m,n) $\left\{\begin{matrix}\text{BINARY}\\\text{DECIMAL}\end{matrix}\right\}$

To group data items into hierarchic structures and structures accessible by array indexing, PL/I provides the following elements:

1) a clause which is used to specify the dimensions for array accessing. It has the form:

   $(m_1, \ldots, m_n)$ for an n dimensional array, where the ith dimension has m_i elements. This clause is used in a DECLARE statement:

   DECLARE data name $(m_1, \ldots, m_n)$ ...

2) a clause which is used to describe hierarchic relationships between data items. It has the same form as the level number clause in COBOL. It is used in a DECLARE statement:

   DECLARE level number data item ...

   level number data item ...

   Such hierarchic structures may also be accessed by array indexing.

For file and storage structures, PL/I provides statements which are used to invoke the DMS access methods of its underlying operating system.

The characteristics of data structures that are made accessible to the programmer by the data description elements of many current higher-level languages are summarized in Section 2.10.

2.8 Data Structures in Data Base Management Systems

Data Base Management Systems are an outgrowth of Information Storage and Retrieval (ISR) systems. ISR systems are designed to manage large quantities of a particular type of data. For example, one early system, MEDLARS, was created to manage documents for the National Library of Medicine (La 1968).

In these systems, since only one type of information was to be used, only one type of file structure was required. Also, input and output routines were specialized to handle the file structure most effectively. As a whole, ISR systems were individually tailored for applications such as text-handling and record-keeping.

The development of more generalized text-handling and record-keeping systems led to today's generalized Data Base Management Systems (DBMS's) (CO 1969).

Every DBMS has a language. The data description statements of the language specify the structure of data maintained by the DBMS. In general, the data description statements form the largest part of a DBMS's language.

For example, in the MARK IV DBMS developed by Informatic Inc. (CO 1969), raw data must be input in the format in which it is to be stored. MARK IV formats can be characterized in the following way.

1. The characteristics of individual data items consist of:
   (i) symbolic naming,
   (ii) the hardware provided character code,
   (iii) fixed lengths as specified by the user,

(iv) data types:

a) character string,

b) number:

1) binary or decimal base,

2) Sign-radix or diminished radix complement (depending on the hardware), and character signs or no signs for decimal numbers,

3) fixed or floating point scale,

(v) data items identified by their position.

2. The characteristics of records consist of:

(i) hierarchic structure,

(ii) fixed order,

(iii) fixed occurrences,

(iv) fixed or varying repetitions ordered as input,

(v) groups of data items identified by their position.

3. MARK IV depends on its underlying Operating System for its storage structures.

MARK IV's language is a tabular language. Forms are provided in which a user selects options provided by the system.

MARK IV is a self-contained DBMS. It is not embedded in any higher-level programming languages. DBMS's which are embedded in some higher-level languages are called host-language DBMS's. They are designed to enhance their host language. This development combines the record structures provided by the host languages with the file and storage structures provided by the DBMS. COBOL and the Honeywell-

General Electric Co.'s Integrated Data Store (IDS) together form an example of this type of system (CO 1969).

In COBOL-IDS, COBOL structures are used at the data item and record level. These structures are described by the standard COBOL statements. The enhancement comes at the file level. IDS adds the capability to describe network relationships among records. IDS networks can be viewed as interconnecting ring structures. The interconnections are maintained by embedded pointers. Each record in IDS may participate in more than one ring. Thus, a single record may be associated with many other records. In each IDS ring there is one record which is treated as a master record. It contains control information. The remaining records in the ring are called detail records. Any record may be master in one ring and detail in another. The data description statements used to describe the characteristics of these rings are in the form of additional clauses in the COBOL record statement. Each ring relationship is defined at level 98 in a record description. In IDS terminology a ring is called a CHAIN. The clause for declaring a record to be a chain master has the form:

98 chain-name CHAIN MASTER.

The clause for declaring a record to be a chain detail has the form:

98 chain-name CHAIN DETAIL
[; SELECT UNIQUE MASTER]
[; MATCH-KEY IS data-name]
[; CHAIN-ORDER IS SORTED]
[; {ASCENDING / DESCENDING} SORT-KEY IS data-name]

[; RANDOMIZE ON data-name]

[; DUPLICATES NOT ALLOWED] .

This clause specifies the chain in which the record is to be a detail, the order in which detail records are to occur (if they are to be ordered), and the field from which a hashed address of the record is to be derived (if this is desired).

MARK IV and COBOL-IDS represent two different classes of DBMS. However, they are both implemented as application programs and are not parts of the operating systems. Many system resources are thus unavailable to the user. Furthermore, privacy protection and access control which are vital to DBMS users are difficult to enforce. Therefore, a different approach to building a DBMS was taken by the designers of the Extended Data Management Facility (EDMF) implemented at the Moore School of Electrical Engineering at the University of Pennsylvania (Ma 1971). The EDMF was implemented as a part of the RCA SPECTRA 70/46 Time Sharing Operating System (TSOS). Statements of the EDMF are in the form of either TSOS Commands, macro-calls which may be used by the regular applications programmer in assembly language programs, or built-in functions for the FORTRAN and COBOL languages.

The set of record and file structures provided by the EDMF are one of the most extensive that has been implemented. EDMF provides record structures which are beyond the COBOL structures (Hs 1971). It provides the following characteristics.

1. The characteristics of individual data items consist of:
   - (i) symbolic naming,
   - (ii) the hardware provided character code,
   - (iii) fixed or variable lengths as specified by the user,
   - (iv) data types:
     - a) character string,
     - b) number:
       - 1) decimal or binary base,
       - 2) sign - radix or diminished radix complement (depending on the hardware) for binary numbers, and character sign or no sign for decimal numbers,
   - (v) value alignment - left for character strings and right for numbers, with zero or blank pad characters,
   - (vi) data items identified by position and by attribute names used as delimiters.
2. The characteristics of records consist of:
   - (i) hierarchic structure,
   - (ii) fixed order,
   - (iii) fixed or optional occurrences of data items and groups,
   - (iv) fixed and variable repetition of data items and groups ordered as input,
   - (v) groups identified by position and by using attribute names as markers.

At the file level, the EDMF allows records to be linked together into lists, when the records contain the same data items (called keywords). A record may be linked into any number of lists. Pointers to the heads of the lists are stored in directories (tables) in ascending lexicographical order. By setting limits on list lengths, files may be implemented completely with pointers embedded in records or with tables of pointers or some combination of the two. This is under the user's control, and allows him to organize his data in a wide range of structures, including inverted, multilist, and indexed random organization (Hs 1970). EDMF seems to be the only existing DBMS to allow the user this kind of control over the implementation of his file.

Each one of the above DBMS's was designed to enhance various characteristics at either the data item, record or file level, or at all three. The level and degree of enhancement vary from DBMS to DBMS. A summary will be provided in Section 2.10 of the most advanced DBMS features.

2.9 The Data Description Language of the CODASYL Data Base Task Group

The CODASYL* Data Base Task Group (DBTG) was organized to unify work done on current DBMS data description languages. The goal of the DBTG is to produce a single data description language (DDL) in which all current data structures at the data item, record and file levels can be described. This DDL (CO 1971) includes:

---

* CODASYL (Conference on Data Systems Languages) is a group originally formed to create a business-oriented language. It produced COBOL and has now extended its interests to DBMS's.

1) the COBOL Data Division which allows the user to specify record formats. Unlike the EDMF, the CODASYL DDL does not allow varying length data items, varying repetitions, or optional occurrences of data items.

2) statements describing network structures. The concept of a SET has been developed to describe file structures. A SET is a sequentially ordered set of records. Each SET has one "owner" record and several "member" records. The concept of "owner" record is similar to that of "master" record in IDS. Member records of SET's are ordered in either of two ways:

(a) Records may be ordered by ascending or descending sequences based on specific keys.

(b) Records may be ordered in relation to existing members of the SET as they are input. That is, when a new record is input, it can be automatically placed as the last or first record of the SET.

The SET concept is similar to the IDS chain.

3) statements describing file implementation. The CODASYL DDL, at the file level, allows the user to specify whether a SET of records is to be implemented either with embedded pointers or with tables of pointers. However, these cannot be combined as in the EDMF, and the user has no control over the pointers or table structure.

In summary, the following characteristics of data structures are made available to the user by the CODASYL DDL.

1. The characteristics of individual data items consist of:
   - (i) symbolic naming,
   - (ii) fixed lengths as specified by the user,
   - (iii) data types:
     - a) character string,
     - b) number:
       - 1) binary or decimal base,
       - 2) sign - radix or diminished radix complement (depending on hardware) for binary numbers, and character sign or no sign for decimal numbers,
       - 3) fixed or floating point scale,
   - (iv) value alignment with blank or zero padding,
   - (v) data items identified by their position.
2. The characteristics of records consist of:
   - (i) hierarchic structure,
   - (ii) fixed order,
   - (iii) fixed occurrences,
   - (iv) fixed and dependent repetitions ordered as input,
   - (v) groups identified by their position.

3. The structure and implementation characteristics of files consist of:

   (i) structuring by input sequence,

   (ii) structuring by criteria on keys (values):

      a) criteria comparisons: $\leq$, $\geq$, $=$,

      b) conjunctions of criteria,

   (iii) implementation:

      a) by embedded pointers,

      b) by tables of pointers.

4. The CODASYL DDL will depend on its implementation for storage structures.

The CODASYL DDL is an attempt to create a common front-end language for describing data structures to DBMS's. There is therefore a degree of overlap between the CODASYL DDL and GDDL developed herein. Before this overlap is discussed, it should be pointed out again that GDDL is designed to be a language for completely describing data structures and for data conversion. The CODASYL DDL is not intended to specify data conversion. Furthermore, GDDL provides the capability of describing storage structures, whereas CODASYL DDL does not. At the record level, CODASYL DDL is based on COBOL and we show in Appendix C that GDDL has more descriptive power than COBOL at the record level. This additional power is obtained by providing more general capabilities for specifying record implementation. At the file level, CODASYL DDL is designed to describe just those file structures

existing in current systems. GDDL is designed to provide much greater descriptive power at the file level. The power is provided by generalizing current file structuring technology essentially by allowing the dependency of file structure on data values, record structure, and record implementation to be described.

2.10 Summary

Two trends have appeared in the handling of data by software systems. First, the data structures provided have become increasingly elaborate, and secondly, the user has been given more and more explicit control over setting up the data structures required.

The earliest systems provided the user with certain structural options at the data item level. These options were, however, provided implicitly through a selection of machine instructions. Successive systems provided more capabilities at the record level, and allowed these to be declared explicitly. It was first in operating systems that structuring facilities were offered at the file level. Typically, the structures provided were limited to a few options which frequently included sequential and indexed sequential structures.

With the development of DBMS's, users were given more control over the implementation and structure of both records and files. However, they still have no control or even knowledge of the storage structures used.

The ddl presented herein takes these two trends towards their logical conclusion. First, the ddl can describe a more general class of data structures than that provided by current data processing technology. Secondly, the ddl allows every aspect of a data structure at each level to be described explicitly.

Those aspects of data structures which have been identified in the preceeding sections have been summarized in Table 2-1. This table is organized to provide a convenient means of evaluating the ddl and its underlying model in later chapters.

Record Characteristics						Machine Languages	Early Operating Systems	Assembly Languages	Early Programming Languages	Current Operating Systems	Current Programming Languages	DBMS's	CODASYL DDL
Data Item Characteristics	Structure Characteristic	Symbolic Naming						X	X		X	X	X
	Implementation Characteristics	Character Code	Fixed by hardware (BCD, EBCDIC, or ASCII)			X		X	X			X	
			User specified (e.g., EBCDIC or ASCII)							X	X		
		Length (precision)	Fixed			X		X	X		X	X	X
			Variable								X	X	
			Dependent									X	
		Value Alignment	Left/Right						X		X	X	X
			Pad Factor						X		X	X	X
		Data Type	Character String			X		X	X		X	X	X
			Number	Base	Binary	X		X	X		X	X	X
					Decimal			X	X		X	X	X
					Octal			X					
				Sign	Unsigned	X		X	X		X	X	X
					Sign Character			X	X		X		X
					Radix or Diminished Radix	X		X	X		X	X	X

Table 2-1. Summary of Data Structure Characteristics

Record Characteristics (continued)						Machine Languages	Early Operating Systems	Assembly Languages	Early Programming Languages	Current Operating Systems	Current Programming Languages	DBMS's	CODASYL DDL
Record Characteristics		Type	Number	Scale	Fixed-Point	X		X	X		X	X	X
					Floating	X		X	X		X	X	X
	Structure Characteristics	Hierarchic Structuring			Depth = 1			X					
					Variable Depth				X		X	X	X
					Array Accessing				X		X		
	Implementation Characteristics	Order			Fixed			X	X		X	X	X
		Occurence			Fixed			X	X		X	X	X
					Variable							X*	
		Repetition			Fixed				X		X	X	X
					Variable							X	
					Dependent						X	X	X
		Repetition Order			As Input				X		X	X	X
					Ascending						X		
					Descending						X		
		Attribute Markers										X*	

* EDMF only

File Characteristics				Machine Languages	Early Operating Systems	Assembly Languages	Early Programming Languages	Current Operating Systems	Current Programming Languages	DBMS's	CODASYL DDL
Structure Characteristics	Structuring by input Sequence				X	Operating System	Operating System	X		X	X
	Structuring by Criteria	Criteria on Values (Keys)	≤					X		X	X
			≥							X	X
			=							X	X
		Criteria on Paths								X*	
		Conjunction of Criteria									X
Implementation Characteristics	Implementing by Sequential Storage				X	Depends on Underlying	Depends on Underlying	X		X	
	Implementation by Pointers	Method	Embedded in Record					X		X	X
			Stored in Table					X		X	X
		Path Length Limit	Upper Bound							X*	

* EDMF only

Structure Characteristics															Storage Characteristics
Blocking														Block Naming	
Printer Pages				Card Decks		Magnetic Disk Packs				Magnetic Tape Reels					
Lines			Page Formatting of Files	Cards	Card Deck Formatting of Files	Cylinders			Disk Formatting of Files	Blocks		Reel Formatting of Files			
Col'ns		Line Formatting of Pages		Card Formatting of Files		Tracks		Cylinder Formatting of Disks		Physical Block Formatting of Reels					
Column Formatting of Lines						Blocks	Track Formatting of Cylinders								
						Block Formatting of Tracks									
															Machine Languages
														X	Early Operating Systems
Depends on Underlying Operating System															Assembly Languages
Depends on Underlying Operating System															Early Programming Languages
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Current Operating Systems
Depends on Underlying Operating System															Current Programming Languages
Depends on Underlying or Associated Operating System															DBMS's
Not Implemented															CODASYL DDL

Storage Characteristics (continued)				Machine Languages	Early Operating Systems	Assembly Languages	Early Programming Languages	Current Operating Systems	Current Program-Languages	DBMS's	CODASYL DDL
Implementation Characteristics	Basic Block Lengths	Tape	Bytes/Block	X	X	Depends on Underlying Operating System	Depends on Underlying Operating System	X	Depends on Underlying Operating System	Depends on Underlying or Associated Operating System	Not Implemented
		Disk	Bytes/Block		X			X			
		Cards	Cards/Deck					X			
		Prin-ter	Columns/ Line					X			
	Labels	Tape	Reels	Start of File Labels		X			X		
				Start of Reel Labels		X			X		
				End of File Labels	X	X			X		
				End of Reel Labels		X			X		
			Blocks	Header Fields		X			X		
		Disk	Blocks	Header Fields		X			X		
		Cards	Decks	Start and End Cards		X			X		
		Prin-ter	Page	Page Headers		X			X		
	Fixed Order of Formatted Device Levels					X			X		
	Fixed Occurrences of Formatted Device Levels					X			X		
	Repetition	Tape	No. of Reels			X			X		
		Disk	No. of Packs						X		
			No. of Cyl's						X		
			No. of Tracks						X		
		Cards	No. of Decks			X			X		
		Pages	No. of Pages			X			X		
			No. of Lines			X			X		

Storage Characteristics (continued)					Machine Languages	Early Operating Systems	Assembly Languages	Early Programming Languages	Current Operating Systems	Current Programming Languages	DBMS's	CODASYL DDL
Mapping Characteristics	Record Positioning Characteristics	Record Split Set	Whole						X			
			Split						X			
		Record-Block Ratio	No. of Records/ Block			X			X			
		Alignment Set	Left/Right					X		X	X	X
			Boundaries	Word				X		X	X	X
				1/2 Word				X		X	X	X
				Byte						X	X	X
			Pad Factor					X		X	X	X
	Pointer Interpretation Characteristics	Mode	Absolute		X		X		X			
			Relative				X		X			
		Type	Main Memory		X		X					
			Device						X			
		Addressing Scheme	Fixed		X		X		X			
		Pointer Form	Cylinder						X			
			Track						X			

CHAPTER 3 RECORD DESCRIPTION

3.1 Introduction

In this chapter we begin our task of showing how the organization of data can be explicitly described. We present the model for record structure that is the foundation for the design of GDDL's record description features. We show that the model is complete for record description in the sense that record structures of Table 2-1 can be described in the model. We also discuss how the model can describe certain generalizations of present record structures. Then we show that the record description statements of GDDL are based on this model. In this way we show that GDDL is also complete and generalized in the above senses. We further demonstrate the completeness of GDDL by noting that the COBOL record description features are properly contained in GDDL and by providing a set of examples which illustrate the ability of GDDL to describe existing record organizations.

3.2 A Model of Record Structures

We begin this section by providing an intuitive introduction to the model.

The smallest meaningful piece of information we will call a "data item". Data items are the components which are organized into records.

Conceptually, a data item is a string of characters, which provide a value for the data item, together with an identification of the

type or class of information to which the value belongs. This type or class of information we call the attribute of the data item.

When a data item is represented on a storage medium, there must be rules which determine how this data item is implemented as a bit string.

When a user is organizing data items for storage and retrieval from a computer medium, he identifies a particular level of organization which is to be stored and retrieved as a single unit when the data is being used. This level of data item organization we call the record level. A convenient way to conceptualize the organization of data items at the record level is as a hierarchy. It is certainly the case that existing software systems (e.g., COBOL, MARK IV, IDS, EDMF, and the CODASYL DDL) provided hierarchies for organizing data items into records. The records are themselves finally represented on a storage medium as a bit string. So again there must be rules for specifying how a particular organization conceived by a user is to be represented as a bit string. There are then the following components to this process of data organization:

for data items:

(1) the conceptual structure of data items,

(2) the encoding of this structure into a bit string, and

(3) the resulting bit string representation;

for records:

(1) the conceptual structure of the records,

(2) the encoding of the record structure into a bit string, and

(3) the resulting bit string representation.

We therefore have to model each of these components. The conceptual structure of data items and records is modelled in terms of the ideas of attribute and value by generalizing the work of (Me 1967), (Ch 1968), (Hs 1970), and (Hs 1971). The bit string is simply a sequence of 0's and 1's. The encoding of the conceptual structure is modelled directly in terms of characteristics for encoding attributes and values as bit strings. The complete model will be presented in two steps. First the model of data items will be described and then the model of records.

3.2.1 The Model of Data Items

3.2.1.1 The Concept of Data Items

The concept of a data item can be described in terms of two primitives - attribute and value, and a definition of data item based on these primitives.

Intuitively, an attribute is a quality, such as size, or weight that is ascribed to an object. For each attribute, there is a set of measures or quantities, known as values. A single value to be associated with the attribute is selected from this set. For example, a measure for the attribute weight is selected from the set of real numbers.

Definition 3-1. A <u>data item</u> is an ordered pair of the form $< a, v >$ where a is an attribute and v is a value.

For example, the pairs < name, JONES >, < age, 32 >, < sex, M >, < school, NEWTOWN HIGH SCHOOL >, < school, UNIVERSITY OF PENNSYLVANIA > are data items.

In representing a data item on a computer medium (such as cards, tape, etc.) both the attribute and the value must be encoded. We shall consider the rules for each kind of encoding separately.

3.2.1.2 Encoding Values

A value is encoded if it is transformed into a bit string according to the following encoding rule. Such a string will be called a value string. The rule for encoding a value is simply a detailed specification of the six characteristics listed below:

1. Character codes. Strings of binary digits are used to encode characters such as letters, numbers and punctuation signs. Character codes have been standardized to the extent that all new computers use either of two codes: USASCII (or ASCII) and EBCDIC. However, it is not sufficient to be able to specify either ASCII or EBCDIC as there are other codes which are in use on earlier computers. Also, users of large data bases employ what are, in effect, new character codes to compress data. Thus, to be completely general, it must be possible to describe any character code. One way to describe a character code is to list for each character the code in terms of its bit string representation.

Associated with a character code is a sort order. To describe the sort order, the characters of the code can be listed in the sort order.

When values are to be translated from one character code to a second character code, it is necessary to indicate for every character in the first code its image in the second code. This can be specified by listing the characters of the second code in the same sort order as the first code.

An example of encoding the characters of a value in EBCDIC is presented below. For the data item < name, JONES >, we have

J → 11010001

O → 11010110

N → 11010101

E → 11000101

S → 11100010

2. Length. The length of a value string is the number of bits in the string.

For example, the value string of the attribute name in the previous example may be specified to be of length 64 bits, where unused bits may be filled arbitrarily.

3. Length Uniformity. If the value strings for an attribute are always of uniform length, then the lengths of the value strings can be described simply by giving the length. However, if the length of value strings for an attribute are not uniform, then either the length of each value string must be given and stored as a data item, or the value string must be delimited by special characters. Thus, value strings may be specified as being either uniform or varying.

4. Value alignment. When the lengths of the value strings for an attribute are to be uniform, the number of characters needed to represent the value may be less than the allotted length. In such cases, it is necessary to specify whether the value is aligned to the right or to the left and to specify the characters to be used to pad out the unused positions.

For example, consider the data item < name, JONES >. The value length of the attribute may have been specified as 64 bits and the character code as EBCDIC. To specify that the value is to be aligned to the left with blank characters used for padding, results in the following encoding of the value JONES:

J → 11010001

O → 11011001

N → 11010101

E → 11000101

S → 11100010

$\not\emptyset$ → 01000000

$\not\emptyset$ → 01000000

$\not\emptyset$ → 01000000

5. Data type. Value strings may be interpreted as either characters or as numbers. Numbers are either signed or unsigned strings of digits. Signs may be denoted by the plus or minus, by radix complement, or by diminished radix complement. Numbers may be organized either as fixed point, or as floating point numbers with the number of significant digits and the length of the mantissa specified.

6. Value criteria. Numeric and set-theoretic criteria may be used to define the set of acceptable values for a given attribute. For example, values of the attribute age may be restricted to numbers between 21 and 65 for a given set of data items. Or values of the attribute city may be restricted to a particular set of city names.

3.2.1.3 Encoding Attributes

We have seen how the value of a data item is encoded. To encode the entire data item we must now provide a way of identifying the attribute to which that value belongs.

This can be achieved in two ways. The first way is to directly encode the attribute as a bit or character string, and then position this string relative to the value. This way of encoding an attribute can be made to fulfill a second role. We saw in the discussion of length uniformity in the section on encoding values, that if a value is specified as having varying length, then it must be delimited by characters which signify the end of the value string. The attribute encoding can serve as such a delimiter for the value string. We will call the string which directly encodes an attribute, an attribute marker.

The following characteristic is used to specify an attribute marker:

7. Attribute marker. Attribute markers can be either character or bit strings which are positioned directly in front of or directly behind a value string.

The second way in which the attribute of a particular value can be identified is by knowing that it always occurs in a certain position relative to other values. That is, if a set of data items are organized in such a way that the position of the value corresponding to a given attribute can be identified, then the attribute has been indirectly encoded by positioning. As the encoding of attributes by positioning depends on the organization of sets of data items, this way of encoding attributes will be discussed in the next section.

3.2.2 The Model of Records

3.2.2.1 The Conceptual Record Structure

In this section we want to model the conceptual structure of records. First, however, we must pin down exactly what we mean by a record itself. Then, we can go on to obtain the structure of such records.

In the data processing field, a user of COBOL conceives of a record differently than say, a user of MARK IV. In the definition of records below, we attempt to give an exact formalization of the notion of record which is independent of any particular software system.

Definition 3-2. A <u>record</u> is a set of data items which are structured according to the following rules:

record → group

group → < attribute, {compound value}>

compound value → compound value, compound value

compound value → group

compound value → data item

We use the symbols < > to denote an ordered set and the symbols { } to denote an unordered set.

For example, the data items < name, JONES >, < age, 32 >, and < sex, M > can be organized into the following record:

< person, {< name, JONES >,
< age, 32 >,
< sex, M >}>

As another example, the data items < name, JONES >, < name, MARY >, < age, 6 >, < name, JOHN >, < age, 10 > can be organized into the record:

< family, {< name, JONES >,
< child, {< name, MARY >,
< age, 6 >}>,
< child, {< name, JOHN >,
< age, 10 >}>}>

In this case < child, {< name, MARY >, < age, 6 >}> and
< child, {< name, JOHN >, < age, 10 >}> are groups.

It should be noted that a data item is simply an attribute-value pair, whereas a group is an attribute-compound value pair. When it is necessary to distinguish the attributes associated with compound values from the attributes associated with values, we will refer to them as group attributes and data item attributes respectively. In the example above, "name" and "age" are data item attributes whereas "family" and "child" are group attributes. Compound values are actually groups or data items. The groups forming a group are called subordinate groups.

We note that as a consequence of the above definition the structure of a record is a hierarchy which has an attribute associated with each part of the hierarchy. We can thus abstract a notion of record structure based on these attributes which is independent of the values. This is done in Definition 3-3.

Definition 3-3. A record structure is a relationship over data item attributes produced according to the following structure productions:

1. record structure → structure
2. structure → < group attribute, {substructure}>
3. substructure → substructure, substructure
4. substructure → structure
5. substructure → data item attribute
6. substructure → null

For example, the data item attributes "name" and "age" may be related by structures obtained from the following structure productions:

family record structure → structure F1

structure F1 → < family, {substructure F1F1}>

substructure F1F1 → substructure F1, substructure F2

substructure F1 → name

substructure F2 → null

substructure F2 → substructure F2, substructure F2

substructure F2 → structure F2

structure F2 → < child, {substructure F21F1}>

substructure F21F1 → substructure F1, substructure F212

substructure F212 → age

Two particular structures of these attributes are:

(i) < family, {name}>

(ii) < family, {name, < child, {name, age} >, < child, {name, age} > } >

Note: i) Production 3 in definition 3-3 allows a particular substructure to repeat an arbitrary number of times, (e.g., in the above example -

substructure F2 → substructure F2, substructure F2).

ii) Production 6 allows the occurrence of a particular substructure to be optional, (e.g., in the above example -

substructure F2 → null).

If we are given a structure, then we can obtain records from it simply by substituting a data item for each data item attribute in the structure.

For example, if we make the following substitutions in the structures above:

< name, JONES >	for name
< name, MARY >	for name
< age, 6 >	for age
< name, JOHN >	for name
< age, 10 >	for age

we obtain the following records:

i) < family, {< name, JONES >}>

ii) < family, {< name, JONES >,

< child, {< name, MARY >, < age, 6 >}>,

< child, {< name, JOHN >, < age, 10 >}>}>

In a previous section we saw how data items were encoded. Now we must consider how the structure of a record is encoded.

3.2 2.2 Encoding the Record Structure

The structure of a record is a relationship over the data item attributes in the record specified by structure productions. These productions actually produce a hierarchic structure which has the data item attributes on the lowest levels and each higher level identified by a group attribute. Therefore, to encode the structure of a record it is only necessary to ensure that the attribute which is associated with each compound value can be identified.

We have seen that the attribute of a data item can be identified by putting a marker adjacent to its value, or, when the data item appears in a group, the attribute can be identified by the position of its value relative to the values of other attributes.

The attribute associated with a compound value can be identified in similar ways. Markers can be placed adjacent to the compound value using the same "attribute marker" characteristic as before. Alternatively, the attribute for a compound value can be identified by the position in which the compound value occurs relative to the compound values of other attributes.

We will now discuss what characteristics must be specified to identify an attribute from the position of the compound value or value. For convenience in this discussion, we will just use the term compound value to refer to both compound values and values.

The attribute associated with a compound value can be identified if the compound value occurs in a particular order with respect to the compound values of other attributes in the same substructure. In this case, the order can be specified by listing the attributes of the compound values in the appropriate order. Further, if one of the attributes in this list corresponds to a substructure which is optional, then it must be specified that this attribute may not appear. Also, if one of the attributes in the list corresponds to a substructure which repeats, then the number of repetitions must be given.

The characteristics required to identify the attribute of a compound value (or value) from the position of the compound value (or value) are given below:

8. Order. The order of compound values can be specified by listing their attributes in the appropriate order. If the attributes are allowed to appear in any order, then the encoding must be done by markers.

9. Occurrence. The occurrence of an attribute may be either mandatory or optional within a substructure.

10. Repetition number. The repetition number is the number of times an attribute may occur consecutively in a substructure.

11. Repetition uniformity. If the number of times an attribute repeats is always the same (i.e., the repetition of the attribute is uniform), then the repetition number can be specified simply by giving the number directly However, if the repetition of the attribute is not uniform, then either the repetition number must be encoded and stored as a data item, or the encoding of the values or compound values for the attribute must be delimited.

12. Repetition order. When the same attribute repeats, then the encoding of the values or compound values for it may either be stored directly in any order or in some order described by criteria on the values.

13. Criteria. Numeric and set-theoretic criteria may be used to define the set of acceptable values or compound values for each attribute.

3.2.3 The Specification of the Encoding Characteristics

In the previous sections we have seen that records are encoded by specifying certain characteristics. We will allow each characteristic to be specified either:

1) directly - by specifying explicitly the characteristic, or
2) indirectly - by specifying a function which must be computed to determine the characteristic. The function may be defined over the values of data items or over other characteristics using the usual arithmetic operators.

For example, the length characteristic can be specified directly as a number of bits, or it can be specified indirectly as perhaps

(i) being equal to the value of some particular data item, or

(ii) being equal to the number of repetitions of some particular attribute.

3.3 Interpretation of Common Data Processing Concepts in Terms of the Model of Record Structures

A set of structure productions together with a specification of the rules for encoding the structures determines a particular type of record, or record type. Two records are of the same record type if and only if they can both be obtained from the same structure productions and they both have the same encoding characteristics.

Note that the term record is sometimes used in data processing literature to refer to what we call a record type.

Note that the production rules of Definition 3-2 make it possible to distinguish easily between a data item and a record consisting of a single data item, even though the both contain a single value. For example, < name, JONES > is a data item, whereas < person, {< name, JONES >}> is a record. This distinction reflects the fact that a data item in itself is only a basic unit of information in some data organization, whereas a data item structured as a record is in addition the basic unit which is stored or retrieved when that data organization is used.

Two groups are of the same group type if and only if they can both be obtained from the same structure productions and they both have the same encoding characteristics.

A data item corresponds to the intuitive idea of a field.

Two fields are of the same field type if and only if they both have the same attribute and are both encoded in the same way.

In early versions of COBOL and in some DMS's only one type of record is allowed per file. In these systems there was therefore no need to refer to particular types of records. However, the model allows for the appearance of more than one type of record in a file. Therefore, some means of referring to particular types of records must be provided. Similarly, it will be useful to be able to refer to particular types of groups and fields. We will use the attribute A of a record (group, field) < A, ... > to name the type of that record (group, field). Thus, a record < person, {...}> is of type person, and a field < age, 10 > is of type age. To ensure that this way of referring to types of records (groups, fields) is unambiguous, we must make the following convention:

Within a file, a given attribute is associated with only one structure and only one set of encoding characteristics.

In particular this requires:

(1) A given attribute can occur in only one production of the form:

structure → < attribute, {substructure}>

(2) If A occurs in a production of the form:

structure → < A, {substructure}>

then A cannot occur in the substructure.

We will see in Section 3.5 that this convention ensures that the structure productions produce only hierarchic organizations.

3.4 An Application of the Model of Record Structures

An example of using the model to completely encode a set of data items in a given structure as a bit string is given below:

Consider the data items - < name, JONES >, < age, 32 >, and < sex, M > and the structure specified by the structure productions:

person record structure → structure P1

structure P1 → < person, {substructure P1P1}>

substructure P1P1 → substructure P11, substructure P1P2

substructure P1P2 → substructure P12, substructure P13

substructure P11 → name

substructure P12 → age

substructure P13 → sex

The following record is obtained from these structure productions:

< person, {< name, JONES >,
< age, 32 >,
< sex, M >}>

The bit string representation of this record is produced using the following encoding characteristics:

(1) The character code for the values of name, age and sex is EBCDIC.

(2) The length of values of name is 64 bits, of age is 16 bits, and of sex is 8 bits.

(3) The lengths of values of name, age and sex are uniform.

(4) The values of name are left aligned and padded with blanks.

(5) The values of name, age and sex are to be interpreted as character strings.

(6) There are no restrictions defined by criteria on the values of name, age and sex.

(7) No attribute markers are used with value strings of name, age and sex.

The structure is encoded according to the following characteristics:

(8) The attributes name, age and sex appear in the order in which they are named by the structure productions.

(9) An occurrence of each attribute is mandatory.

(10) Each attribute occurs once in a structure.

(11) The repetition for each attribute is uniform.

(12) Since there may be only one occurrence of the attributes name, age and sex, the repetition order criterion does not apply.

(13) There are no restrictions defined by criteria on the compound values of person.

Applying these encoding characteristics, the following record representation results:

11010001110101101101010111000101111000100100000001000000010000000
1111001111111001011010100

For every different set of data items which are substituted in the structure obtained from the above set of structure productions, a different bit string is produced by these encoding characteristics.

3.5 The Completeness and Generality of the Model

To be complete, the model must incorporate in itself all of the characteristics of record structures derived in Table 2-1. This is done for the data item characteristics as follows:

Symbolic naming appears in the model as the concept of an attribute.

The implementation characteristics for data items appear in the model directly as encoding characteristics.

The characteristics relating to the structure of records are incorporated in the model as follows:

The structuring characteristics of records appear in the model as the concept of record structure.

The implementation characteristics are incorporated directly as encoding characteristics.

Thus, the model includes each of the record level characteristics appearing in Table 2-1. In this sense, the model is complete.

We further note that the structure productions and the convention of Section 3.3 impose a partial ordering on the attributes of a structure. This is proved as follows:

Theorem: The structure productions and the convention of Section 3.3 impose a partial ordering over the attributes of a record structure.

Proof: A partial ordering is a relation which is

1) reflexive,
2) antisymmetric
3) transitive.

Let us define $\supseteq$ to be a relation over attributes as follows: for attributes a and b, $a \supseteq b$ if and only if a = b, or < a, { ... b ... }> is a structure, where b may appear in any depth of { , } or < , > brackets.

We will now show $\supseteq$ is a partial ordering.

1) By definition $\supseteq$ is reflexive.

2) Assume that $a \supseteq b$ and $b \supseteq a$ and that $a \neq b$.

This means < a, { ... b ... } > and < b, { ... a ... } > are structures. But by (1) of the convention, the attribute b can only be associated with one substructure which must therefore be { ... a ... }. Thus, < a, { ... b ... } > is actually < a, { ... < b, { ... a ... } >. This is not allowed by (2) of the convention. Thus, $a \supseteq b$ and $b \supseteq a$ implies a = b. Hence $\supseteq$ is antisymmetric.

3) Assume $a \supseteq b$ and $b \supseteq c$.

If a = b and/or b = c, then $a \supseteq c$.

If < a, { ... b ... } > and < b, { ... c ... } > are structures, then by (1) of the convention < a, { ... b ... } > is actually < a, { ... < b, { ... c ... } > ... } >. Thus, $a \supseteq c$, and $\supseteq$ is transitive.

Therefore, $\supseteq$ is a partial ordering.

Mathematically, any hierarchy can be realized by a partial ordering (Bi 1948). From the above proof, it follows that the structure productions and conventions can realize any hierarchic record structure.

The characteristics of Table 2-1 are incorporated in more generalized forms in the model to allow for the description of variations of existing data structures. This generality is provided in

the following ways:

1) The model provides a more generalized way to describe the order of data items and groups. As we have seen in Table 2-1, current systems only provide for the specification of fixed ordering. However, the ordering characteristic of the model allows order to be specified as fixed or as arbitrary relative to the groups. For example, consider the following group -

$$< x, \{ < y, \{ < z, a >, < t, b > \} >,$$
$$< u, c >,$$
$$< v, \{ < w, d >, < s, e > \} > \} >$$

with the following order characteristic:

The ordering for the compound values of attribute x is fixed, and the ordering for the compound values of attributes y and v is arbitrary.

This results in the following valid orderings of the values a, b, c, d, e:

abcde, bacde, abced, and baced.

Such variable orderings are not permitted in current systems.

2) The model provides a more generalized way to specify the encoding characteristics than is required to describe the characteristics of Table 2-1.

In Table 2-1, we saw that the characteristics length and repetition could be specified as depending on some single other data item. In the model, all characteristics can be specified as depending on other data

items, other characteristics and functions of these. This greatly increases the variety of encodings which can be specified.

In these ways, the model allows generalizations of current data representations at the record level to be specified.

3.6 The Relationship Between the Model and GDDL

GDDL has been explicitly designed in terms of the model. A GDDL statement consists of an identifying name and a string of parameters. The FIELD and GROUP statements are used to describe the conceptual organization of data items and groups. Each encoding characteristic of data items and the structure of records can be specified by one or more parameters in GDDL statements. The parameters and statements for these characteristics are listed in Table 3-1 given below:

Value Characteristics	Statements and Parameters	Remarks	Specified in Section in Appendix A
Character Code	FIELD statement parameter (ii) CHAR statement SET statement	Data item Characteristics	1.1 1.4.1 2.1.2.1
Length	FIELD statement parameter (iii) parameter (iv)		1.1
Length Uniformity	FIELD statement parameter (v)		1.1
Value Alignment	FIELD statement parameter (ix)		1.1
Data Type	FIELD statement parameter (vi)		1.1
Value Criteria	GROUP statement parameter (iii)f Criterion statements		1.2 2.1

Attribute Characteristics	Statements and Parameters	Remarks	Section in Appendix A
Attribute Markers	CONCODE statement	Record Characteristics	1.4.3
Order	GROUP statement parameters (ii) and (111)a		2.1
Occurrence	GROUP statement parameter (iii)b		2.1
Repetition Number	GROUP statement parameter (iii)c		2.1
Repetition Uniformity	GROUP statement parameter (iii)d		1.2
Repetition Order	GROUP statement parameter (iii)e		1.2
Criteria	GROUP statement parameter (iii)f		1.2
	Criterion statements		2.1
Specification of Characteristics	Statements and Parameters		Section in Appendix A
Direct	By listed parameters		
Indirect	Parameter statements		1.4.4

Table 3-1. The Relationship Between the Model and GDDL

Insight into the relationship between the model and GDDL can best be obtained by comparing the format of the GDDL FIELD and GROUP statements with the definitions of field type and group type (see Section 3.3 and Definitions 3-1 and 3-3).

The FIELD statement has the following format:

FIELD (field name, encoding characteristics)

This corresponds to the specification of a field type in the following way. The attribute corresponds to the field name, and encoding characteristics appear directly. Thus, we see that the FIELD statement specifies data items.

The GROUP statement has the following format:

GROUP (group name, ... ; (list), ... , (list) ...).

This corresponds to the specification of a group type in the following way. Compare the structure productions of Definition 3-3 with this format. The production of the type:

structure → < attribute, { substructure } >

corresponds to the format of the GROUP statement, with the attribute corresponding to the group name, and with all the substructures that can be obtained using the remaining types of productions corresponding to (list), ... , (list). The encoding characteristics for each substructure are included in each list. Thus, we see that the GROUP statement specifies the structure for groups. To specify that a particular group is to be treated as a record, the RECORD statement is used (see Section 1.3 in Appendix A).

From the above table, we note that every characteristic of the model is included in CDDL. Since the complete set of characteristics can encode the structure and values of data items, CDDL therefore has the same capability. This, in effect completes the argument that CDDL can specify any record level structures which can be described in

the model.

3.7 Demonstrations of GDDL's Completeness

In the previous section we showed that GDDL is complete for record description by showing that the model on which it was based is complete. We now provide several practical examples of its completeness.

The first of these examples is a demonstration that GDDL contains the COBOL record description features as a proper subset. COBOL was chosen because it is the prototype for almost every DBMS DDL and for the CODASYL DDL effort. It has the most highly developed record description capabilities currently available. The demonstration is given in Appendix C, part 1. In Appendix C, part 2 three examples are given of record characteristics describable in GDDL but not in COBOL.

The remaining examples demonstrate the use of GDDL in describing real-world records. These record descriptions are part of larger examples of complete conversions of data from one structure to another. They are given in Appendix B.

CHAPTER 4 FILE DESCRIPTION

4.1 Introduction

This chapter is devoted to the study and description of organizations of records called files. We develop a model of file structures which is a very general extension of current concepts of files as analyzed in Chapter 2. This model leads to the technique for describing file structures that is incorporated in GDDL. This technique is illustrated in a series of examples which show that GDDL can describe several well-known file structures.

4.2 A Model of File Structures

In Chapter 3, we developed a model of records. In this chapter, we are concerned with the record as a basic unit of storage and retrieval. When large numbers of records are to be stored and retrieved, a problem of efficient utilization arises. For example, store time is conserved if data need not be rearranged each time a new record is stored. And search time is conserved if records can be so arranged that each record is stored physically next to the record that is needed next. Then, when the first record to be used is found, succeeding records can be directly accessed in the order of usage. However, when access to two or more records from a single record is required, a sequential ordering of records does not in itself provide the most efficient utilization.

A user, then, should conceive of the records as being connected together in some way by access paths. These paths make a record at

one point on a path accessible to records which occur at points previous to it on the path. They represent connections among the records in question that the user wants to exploit for storage and retrieval. We call such an organization of records the conceptual file structure. When this structure is implemented on a storage medium, it must be represented in some way by a string of bits.

As seen in Chapter 2, there are currently three ways in which the access paths of a file structure are implemented. If there is to be an access path from a record (say, A) to another record (say, B), it may be implemented by:

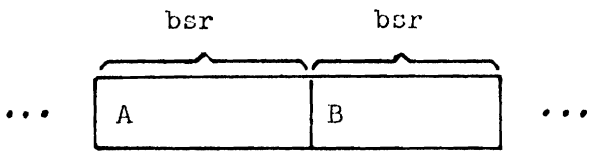
(1) sequencing position - the bit string representation for B is concatenated after the bit string representation for A (see Figure 4-1, a);

(2) embedding pointers in the records - a pointer to B (i.e., an encoding of the position that the bit string representation of B occupies in the record sequence) is included as a field in A (see Figure 4-1, b);

(3) arranging pointers in tables - a pointer to B is concatenated after the pointer to A in a sequence of pointers (called a table) which is maintained separately from the records themselves.

Ultimately, a pointer to B will give the physical address of the bit string representation of B when it is stored on a storage medium. How the actual bit string for a pointer can be obtained is discussed in

Chapter 5, after we have considered the organization of storage media.



where bsr means: bit string representation of

Figure 4-1, a. By Sequencing

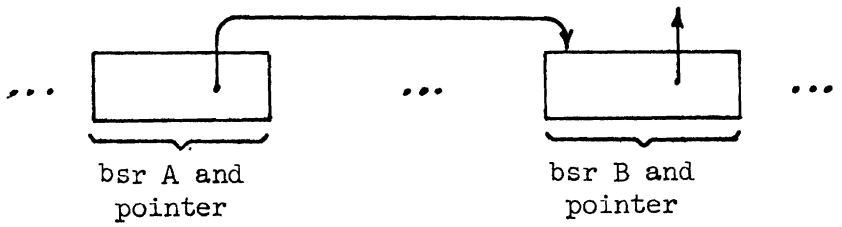


Figure 4-2, b. By Embedding Pointers

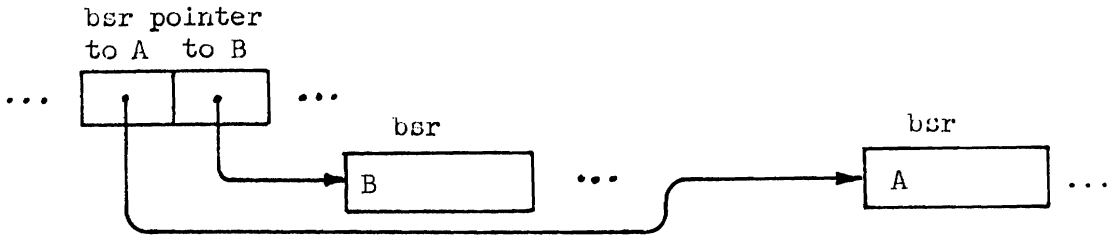


Figure 4-1, c. By Using Tables of Pointers

Figure 4-1. Implementation of Access Paths

We saw in Chapter 3 how the records themselves are encoded as bit strings. Now we must consider the rules for encoding the file structure into a bit string. If the file structure is to be implemented by sequencing, the rules must determine the sequence in which the bit strings representing the records occur. If the file structure is to be implemented by pointers, the rules must determine how the pointers are encoded into bit strings, where these bit strings must be positioned in relation to the bit strings of the records, and the sequence in which the bit strings of the records must occur. These rules will then determine a bit string which represents the file structure.

There are thus three components of this process:

(1) the conceptual file structure,

(2) the final bit string, and

(3) rules for encoding the conceptual file structure of records as a bit string.

We therefore have to model each of these components. The modelling of the conceptual structure is influenced by (Co 1970). The rules for encoding are modelled after the work of (Hs 1970). The bit string is simply a sequence of 0's and 1's.

First, the conceptual file structure will be described. And secondly, the rules for encoding the file structure will be specified.

4.2.1 The Conceptual File Structure

We noted in the previous section that the file structure determines which records are connected by access paths. In other words, it

determines a relation (called a file relation) among records on the basis of access paths. Consider two records which we will call A and B, such that either

(i) the bit string representation of B is concatenated after the bit string representation of A, or

(ii) there is a pointer from A to B.

Then we say that there is a direct access path from A to B. Relative to this path we call record A the head of the path and record B the tail of the path. This terminology allows us to refer to records connected by access paths without naming the specific records.

Definition 4-1. The file relation determined by access paths through a set of records consists of the set of ordered pairs < head record, tail record > for each direct access path.

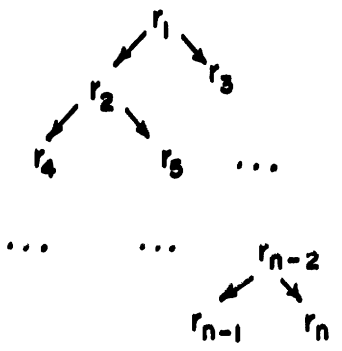
As examples, consider that we are given a set of records, $S = \{r_1, \ldots, r_n\}$ where r_i is a record for $1 \leq i \leq n$.

(1) The access paths of the list structure:

$r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots r_{n-1} \rightarrow r_n$

give the relation $I_1 = \{ < r_1, r_2 >, < r_2, r_3 >, \ldots , < r_{n-1}, r_n >\}$.

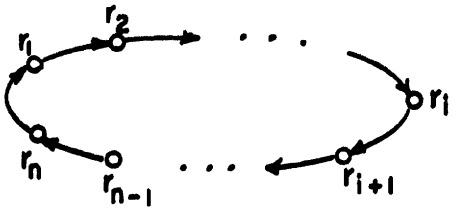
(2) The access paths of the tree structure:



give the relation $I_2 = \{< r_1,r_2 >, < r_1,r_3 >, < r_2,r_4 >,$

$< r_2,r_5 >, \ldots , < r_{n-2},r_n > \}$,

(3) The access paths of the ring structure:



give the relation $I_3 = \{< r_1,r_2 >, \ldots,$

$< r_i,r_{i+1} >, \ldots, < r_{n-1},r_n >, < r_n,r_1 >\}$

It will be convenient to introduce the following terminology:

a) If the pair of records $< r_i,r_j >$ is in a file relation R, then we say that there is a <u>path of length 1</u> from r_i to r_j for relation R. Therefore, a direct access path has length one.

b) If the pair of records $< r_i,r_j >$ is <u>not</u> in a file relation R, we say there is a <u>path of length 0</u> from r_i to r_j for relation R.

c) If the pairs of records

$< r_1, r_2 >, < r_2, r_3 >, \ldots, < r_i, r_{i+1} >, < r_{i+1}, r_{i+2} >, \ldots,$

$< r_n, r_{n+1} >$

are in a file relation R, then we say that there is a <u>path of length n</u> from r_1 to r_{n+1} for relation R.

To model the conceptual file structure we must have a way to specify any file relation that a user may require. In general, there may be an arbitrarily large number of records that can be included in a file structure. Therefore, it is not practical for a user to state the file relation extensively by listing all the pairs of records. Instead, he can specify criteria over the records which will determine when two such records are to be in the relation. Thus, for two records A and B, < A, B > is a member of a file relation if and only if A and B satisfy the criteria for the relation. Such criteria can describe explicitly the conditions which must be met for two records to be connected by a direct access path.

We provide below a set of production rules for specifying criteria.

At this point it is worth noting that in Chapter 3, we were only concerned with hierarchic organizations and so simple production rules were all that was necessary to specify record structures. However, to organize records into files, a far wider variety of organizations is required and, therefore, a more elaborate way of specifying them is necessary.

Definition 4-2. A file structure is a file relation determined by criteria obtained from the following production system:

Criterion Production System:

Primitives: attribute, bit string, character string, characteristic, integer, arithmetic relations (=, ≤, etc.), arithmetic operators (+, -, etc.), set membership relation (ε)

Rules to produce the names of records, fields, characteristics and paths:

index → (integer)

record-modifier → HEAD
→ TAIL
→ X integer

attribute-form → attribute
attribute index

record-attribute → attribute record-modifier
→ attribute

attribute-modifier → attribute-form
→ attribute-form OF attribute-modifier
→ attribute record-modifier

record-reference → record-attribute
→ record-attribute criterion

field-reference → attribute-modifier

characteristic-reference → characteristic

path-reference → PATH (record-reference, record-reference, criterion)

Rules to produce set-theoretic criterion:

constant → character string

→ bit string

set-member → field-reference

→ constant

→ set-member, set-member

set → {set-member}

set-criterion → field-reference ϵ set

→ characteristic-reference ϵ set

Rules to produce arithmetic criterion:

term → VALUE (field-reference)

→ PARAMETER (characteristic-reference)

→ LENGTH (path-reference)

→ constant

arithmetic-operator → * → - → x → /

relation-symbol → = → ≠ → < → ≤ → > → ≥

arithmetic-expression → term

→ (arithmetic-expression) arithmetic-operator (arithmetic-expression)

arithmetic-criterion → arithmetic-expression relation-symbol arithmetic-expression

Rules for quantifying and combining criteria:

quantifier → ALL (X integer)

→ SOME (X integer)

criterion → arithmetic-criterion

→ set-criterion

→ quantifier (criterion)

→ NOT (criterion)

→ (criterion) AND (criterion)

→ (criterion) OR (criterion)

Note: A quantifier is required in a criterion only when a record-modifier contains a string of the form: X integer. There must be one quantifier of the form: ALL or SOME (X integer) for each unique string X integer in the criterion.

This production system is used to specify criteria which determine when there is to be a direct access path from one record to another.

We imagine a processor which is compiling a file relation over a set of records. For any two records that the processor picks up (potential head and tail records), we describe to the processor criteria which determine whether the two records are to be linked. The criteria can be over:

(i) the values of data items in the records,

(ii) the structural properties of the records,

(iii) the implementation of the records, and/or

(iv) any linkages already compiled.

The first three factors produce data and record dependent file structures, and the last factor produces purely graph-theoretic structures. These criteria are expressed as arithmetic and set-theoretic expressions. The values and characteristics being tested in criteria may occur in head or tail records or in any number of distinct records other than the head or tail records; and direct access paths may similarly exist between head, tail and arbitrary records. When criteria are specified for records other than head or tail records, record-modifiers of the form X integer are used in referring to these records. For each unique reference of this kind, there must be a quantifier which indicates whether the criterion in which the reference appears must hold for all, or at least one, of the records of the type in question.

As examples, consider the following set of records:

$S = \{r_1 = <$ person, $\{<$ name, JOHN DOE $>,$
$<$ soc. sec. no., 073028556 $> \} >,$

$r_2 = <$ person, $\{<$ name, JAMES DOE $>,$
$<$ soc. sec. no., 029110076 $>,$
$<$ spouse, MARY BROWN $>,$
$<$ child, JOHN DOE $> \} >,$

$r_3 = <$ person, $\{<$ name, MARY BROWN $>,$
$<$ soc. sec. no., 008412637 $>,$
$<$ spouse, JAMES DOE $>,$
$<$ child, JOHN DOE $> \} >,$

r_4 = < person, {< name, MARK BROWN >,
< soc. sec. no., 214325629 >,
< spouse, ALICE BROWN >,
< child, MARY BROWN > } >,

r_5 = < person, {< name, ALICE JONES >,
< soc. sec. no., 345291102 >,
< spouse, MARK BROWN >,
< child, MARY BROWN > } > }

(1) Consider now a criterion which orders the records by soc. sec. no. The criterion which determines when there is a direct access path from a HEAD record of type person to a TAIL record of type person can be stated in English as:

> The value of soc. sec. no. in the HEAD record is less than the value of soc. sec. no. in the TAIL record; and there is no other record of type person having a soc. sec. no. between the one in the HEAD record and the one in the TAIL record.

This criterion is expressed using the Criterion Production System as:

> (soc. sec. no. OF person HEAD < soc. sec. no. OF person TAIL) AND (ALL (X1) (NOT (soc. sec. no. OF person HEAD < soc. sec. no. OF person X1) AND (soc. sec. no. OF person X1 < soc. sec. no. OF person TAIL)))

This determines the following file structure:

$r_3 \rightarrow r_2 \rightarrow r_1 \rightarrow r_4 \rightarrow r_5$

Note: In this example - soc. sec. no. OF person HEAD

soc. sec. no. OF person TAIL

soc. sec. no. OF person X1

are all value-references.

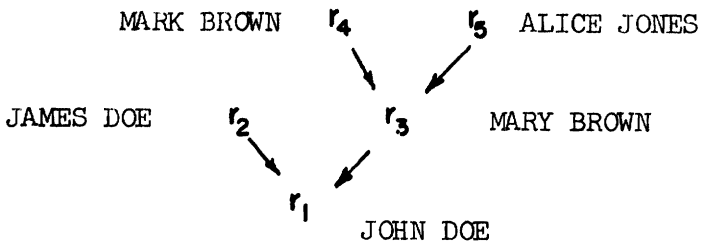
(2) Consider a criterion, which arranges the records into a family tree. The criterion which determines when there is a direct access path from a HEAD record of type person to a TAIL record of type person can be stated in English as:

The value of name in the HEAD record equals the value of child in the TAIL record.

This criterion is expressed using the Criterion Production System as:

name OF person HEAD = child OF person TAIL

This determines the following file structure:



Note: In this example, name OF person HEAD and child OF person TAIL are value-references.

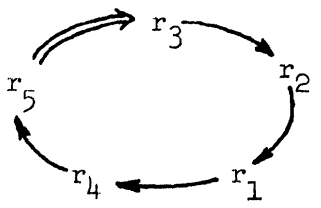
(3) Finally, consider a criterion which, combined with the criterion of example (1), creates a ring structure. The criterion which determines when there is a direct access path from a HEAD record of type person to a TAIL record of type person can be stated in English as:

> There is a path of any length determined by the criterion of example (1) from the TAIL record to the HEAD record; and there is no path determined by the criterion of example (1) either from the HEAD record or to the TAIL record.

This criterion is expressed by the Criterion Production System as:

> (LENGTH (PATH (person TAIL, person HEAD, criterion (1))) ≥ 1) AND (ALL (X1)((LENGTH (PATH (person HEAD, person X1, criterion (1))) = 0) AND (LENGTH (PATH (person X1, person TAIL, criterion (1))) = 0)))

where criterion (1) is the criterion of example (1). This, together with criterion (1), determines the following file structure:



where access paths of the form → are determined by the criterion of example (1) and access paths of the form ⟹ are determined by the above criterion.

In representing a file on a computer medium both the records and the file structure must be encoded. The records are encoded into bit strings as discussed in the previous chapter. The next section discusses the encoding of the file structure.

We shall consider each of these encoding characteristics separately.

1. Encoding method. The encoding method is either sequential, embedded pointer, or pointer table encoding.

2. Link number. The link number is the number of tail records to which any head record is connected by a direct access path. In encoding embedded pointers, this is the number of pointers that may be stored in a single head record to encode a particular file relation (or the maximum such number). In pointer table encoding, the link number is the number (or maximum number) of entries in the table for a single head record.

3. Linkage uniformity. If the link number is to be always the same for each head record (i.e., the number of access paths starting from any record is uniform), then the link number gives the actual number of those access paths. Otherwise, the link number gives an upper bound on the actual number.

4. Path length. Path length gives an upper bound on the length of a path encoded by embedded pointers. For example, for a tree structure, path length sets the maximum depth of the trees. If the maximum is reached and more records remain to be connected by paths, a new structure containing these records is started.

5. Connection set number. The connection set number gives the maximum number of records that are connected together by access paths. For example, in a tree structure the connection set number gives the maximum number of records in the tree. If the maximum is reached and more records remain to be connected by paths, a new structure is started.

4.2.2 Encoding the File Structure

When a file structure is encoded, it is the actual ordered pairs in the file relation that are encoded and not the criterion determining the file relation. It is understood that the ordered pairs of the file relation have been compiled first from the criterion by some mechanical process. The user, however, is only required to supply the criterion.

A file relation can be encoded in either one or in a combination of the three ways discussed above. That is:

(1) sequential encoding - the bit string representing the tail record is concatenated after the bit string representing the head record;

(2) embedded pointer encoding - a pointer to the tail record is encoded as a value in the head record; and

(3) pointer table encoding - a pointer to the tail record is concatenated after a pointer to the head record in a table.

The particular method is chosen to optimize the processing of data.

The rules for encoding file relations are simply a detailed specification of certain characteristics that determine the above three encoding methods. The characteristics required to completely specify each encoding method are listed in Table 4-1.

Encoding Method	Required Characteristics
Sequential	1
Embedded Pointers	1,2,3,4,5
Pointer Table	1,2,3

Table 4-1. Characteristics for Each Encoding Method

This now completes the characteristics which must be specified to determine the rules for encoding a file relation.

As for the encoding characteristics for record description, we allow each characteristic to be specified either:

1) directly - by specifying explicitly the characteristics, or

2) indirectly - by specifying a function which must be computed to determine the characteristic. The function may be defined over the values of data items or other characteristics using the usual arithmetic operators.

For example, the link number characteristic can be specified directly, or it can be specified indirectly as, perhaps, being equal to the value of some data item in the head record in question.

We will now show how these characteristics are used to encode a file using the embedded pointer or pointer table methods:

(1) When using either of these methods to encode a file, another criterion must be defined to determine the sequence of the records.

(2) When a file relation is encoded by embedded pointers, a field is specified at the record level to contain the pointers. This field forms a part of the actual record and is itself encoded in the usual way.

(3) When a file relation is encoded by pointer tables, the encoding of the table must also be described. This is done by treating the table as a file whose records contain the pointers.

4.3 Applications of the Model of File Structures

We will now illustrate how the model is used to encode a file structure into a bit string. We will take the file structure specified in Example (1) and encode it into a bit string for each of the sequential, embedded pointer and pointer table methods.

(1) Sequential encoding.

The characteristic specified below:

1. encoding method is sequential;

applied to the file structure results in the bit string illustrated in Figure 4-2.

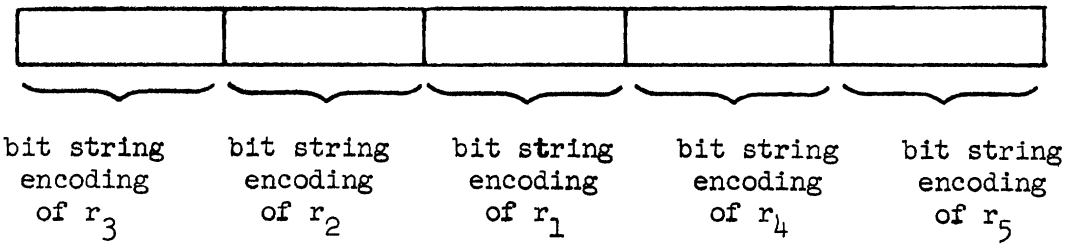


Figure 4-2. Bit String Representation of File Sequentially Encoded

(2) Embedded pointer encoding.

Let us assume that a field has been specified at the record level that is to contain the pointer. Assume this field is positioned at the end of the person record. Let us also assume that the record sequence is to be arbitrary. Then the following characteristics applied to the file structure of Example (1) results in the bit string illustrated in Figure 4-3.

1. encoding is by embedded pointer,
2. the link number is 1,
3. the link number is uniform,
4. no limits are put on path length, and
5. no limits are put on the number of records linked together.

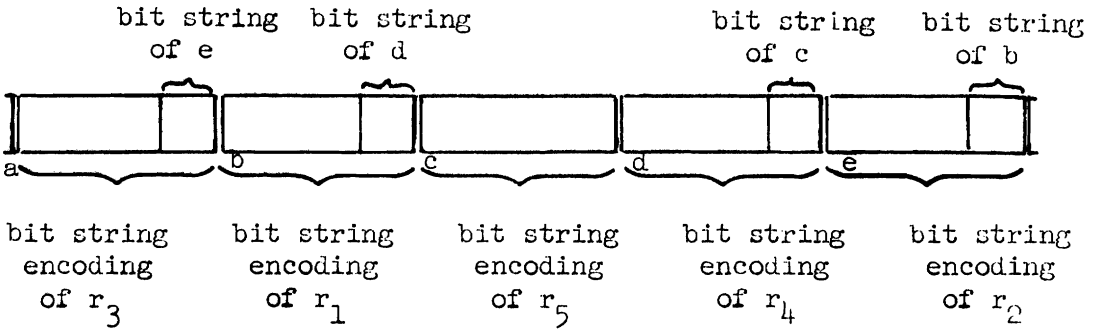


Figure 4-3. Bit String Representation of File Encoded by Embedded Pointers

In this bit string the order of the records is r_3, r_1, r_5, r_4, and r_2. a, b, c, d, and e are the positions of each record in the string. These appear in each record as pointers. This is illustrated in Figure 4-4.

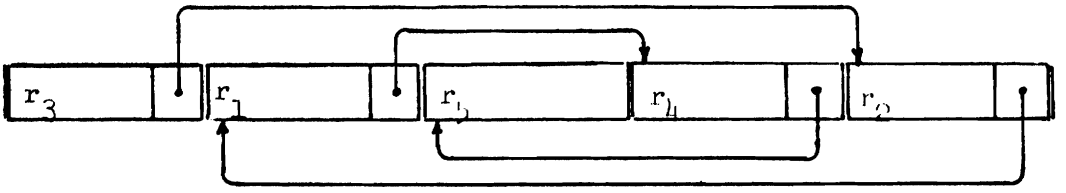


Figure 4-4. File Linked by Embedded Pointers

(3) Pointer table encoding.

Let us assume that each record of the table is to contain only a single field for each pointer and that the sequence of the person records is to be arbitrary. The following characteristics specifying the organization of person records and the organization for records of the table, applied to the file structure of Example (1) results in the bit string illustrated in Figure 4-5.

a) table specification -

1. encoding of the table is sequential.

b) file specification -

1. encoding is by a pointer table,
2. the link number is 1,
3. the link number is uniform.

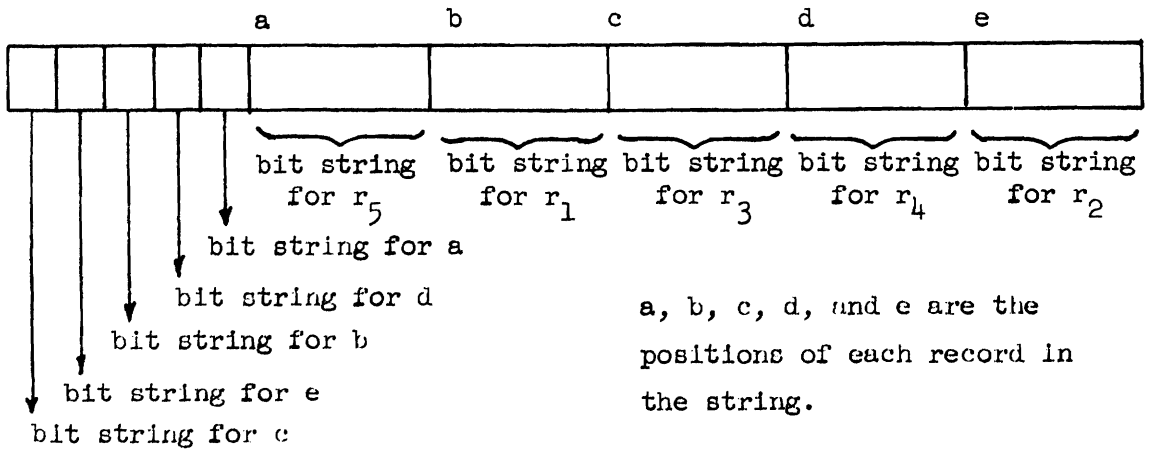


Figure 4-5. Bit String Representation of File Encoded by a Pointer Table

The first five bit strings represent pointers to the records, as illustrated in Figure 4-6.

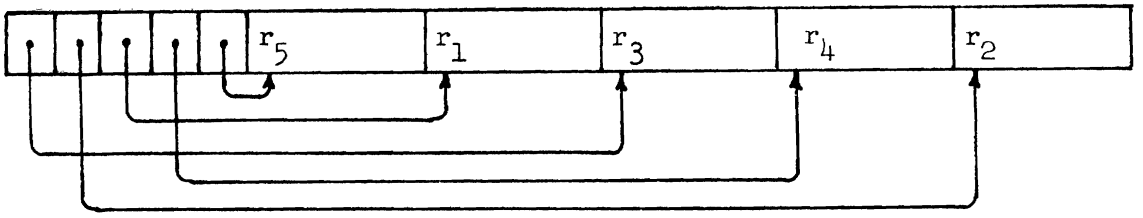


Figure 4-6. File Linked by a Pointer Table

4.4 The Completeness and Generality of the Model

We will now show that the model of file structures is complete. We will do this by showing that the structuring characteristics at the file level of Table 2-1 are properly contained in the conceptual part of the model and that the implementation characteristics are contained in the encoding rules of the model. The structuring characteristics of each software system are actually arranged to provide a highly restricted means for specifying criteria for connecting records. The criterion production system for the model allows the criteria specifiable by each software system to be expressed. The implementation characteristics of every software system as derived in Table 2-1 are specified directly by the encoding characteristics of the model. Thus, every file that can be specified in the software systems of Chapter 2 can be specified in the model. In this sense, the model is complete.

We will now show how the model is generalized.

The criterion production system which is the basis of the conceptual part of the model is more general than any means of specifying criteria by an existing software system in three ways:

(1) it allows criteria to be defined on the actual encoding characteristics, as well as on values or paths,

(2) it allows criteria to be defined over arithmetic and set-theoretic functions of values, characteristics and paths, and

(3) it allows a criterion to be defined in terms of more than one criterion where the criteria are connected by the connectives: AND, OR and NOT.

The use of the criterion production system (cps) to describe file structures is intended to avoid the deficiencies inherent in certain other approaches. We first note that we reject the implicit approach, where file structures are given implicitly by programs used to access data in the structures, because such programs tend to obscure rather than emphasize the logical and graphical basis of the file structure. We need to describe file structures so that they are easily understood by humans as well as machines. A second, and apparently straightforward, approach would be to name explicitly what linkages are required in terms of common graphical structures such as trees and rings. This approach is inadequate for two reasons. First, there are many ways in which such linkages are determined. For example, the linkages between records in a sequential structure may be determined on the basis of the values occurring in some particular type of field, or they may be determined by

a particular record encoding characteristic such as the length of a field. It is inadequate to specify a type of file structure without specifying what properties of records are used to determine that structure. Secondly, there is no capability for describing new file structures. A third approach, which lies at the opposite extreme to the one above, was suggested but not provided in (Co 1970). Codd suggests using an applied predicate calculus to define how linkages are constructed from certain primitive relations and functions. This approach provides generality, but does not provide a description which is immediately intelligible to a user in terms of accepted data processing concepts such as field, record and file. Furthermore, the predicate calculus alone does not provide the methods for encoding the data and the means for implementing the description.

Our approach in the cps is to provide as much of the explicit information provided by the second approach as possible while preserving the generality inherent in the third approach. The primitives of the cps include only data-related concepts like attribute and characteristic and arithmetic relations like "greater than" and "not equal to". Criteria produced by the cps are intelligible and refer explicitly to relations over records, values and characteristics. This can be seen by comparing the English and cps ways of expressing criteria in the examples given after Definition 4-2. The cps expressions largely retain the intuitive concepts of the English expressions. The cps has quantifiers like the ones in the predicate calculus, but these quantifiers apply only to predicates which are intuitively meaningful criteria. The concept of

variable appears as a general name for a record to be used when all or some records are to be tested for a particular property. It is not a variable in the algorithmic sense. In summary then, the cps allows data structure terminology to be used explicitly in a very general way of describing file structures.

The implementation methods included in the model are a generalization of the work of (Hs 1970). In this paper, Hsiao shows how different common file structures are special cases of a generalized way of implementing files using tables of pointers (called directories) and embedded pointers to create access paths between records. The generality of this work is increased by extending it to include the sequencing of records.

Finally, the model provides a more generalized way to specify encoding characteristics than is necessary for describing the characteristics of Table 2-1. The encoding characteristics for file description as well as those for record description can be specified as depending on data items, other characteristics and functions of these. This greatly increases the variety of encodings which can be specified.

In these ways, the model allows generalizations of current file structure technology to be described.

4.5 The Relationship Between the Model and GDDL

GDDL has been explicitly designed in terms of the model. The LINK and criteria statements are used to describe the conceptual file structure and the LINK and FILE statements are used to describe the implementation of files. The exact relationship between GDDL and the model is described by Table 4-2.

Conceptual Part of the Model	Statements and Parameters	Remarks	Section in Appendix A
Criterion Production System (CPS)	LINK Statement	used to name the head and tail records, and the criterion for linking them	2.2
	Criterion Statements		2.1
Encoding Characteristics	Statements and Parameters		Section in Appendix A
Encoding Method	LINK statement parameter (v)		2.2
Link Number	LINK statement parameter (vi)		2.2
Link Uniformity	LINK statement parameter (vii)		2.2
Path Length	LINK statement parameter (v)		2.2
Connection Set Number	LINK statement parameter (v)		2.2
Specification of Characteristics	Statements and Parameters		Section in Appendix A
Direct	By listed parameters		
Indirect	Parameter statements		1.4.4

Table 4-2. The Relationship Between the Model and CDDL

This completes the argument that GDDL can describe any file representation which can be specified in the model. Therefore, GDDL is complete and general in the sense described above.

4.6 Demonstrations of GDDL's Completeness

In the previous sections we showed that GDDL is complete for file description by showing that the model on which it was based is complete. We now provide practical examples of its completeness. These examples demonstrate the use of GDDL in describing real-world files. The file descriptions are part of larger examples of complete conversions of data from one structure to another. These examples are given in Appendix B.

CHAPTER 5 STORAGE DESCRIPTION

5.1 Introduction

In this chapter we complete our task of showing how the organization of data can be explicitly described. We now present a final model to show how a bit string representation of a file, produced by the models of Chapters 3 and 4, is transformed to its final physical representation (e.g., a sequence of appropriately positioned magnetic spots on a tape). This model of storage structures is shown to be complete by demonstrating that it incorporates all of the storage level characteristics derived in Table 2-1. We also discuss how the model includes certain generalizations of some of these characteristics. Then we show that the storage level GDDL is based on this model. In this way we can show that GDDL is also complete and generalized in the above sense. We further demonstrate the completeness of GDDL by providing a set of examples which illustrate the ability of GDDL to describe data structures on particular devices.

5.2 A Model of Storage Structures

In the previous chapters we followed the process by which a user organizes his data into records, encodes these records as a bit string, then sets up access paths between records and finally encodes these access paths to produce a single bit string representing his data. Now the user must specify how this bit string is to be distributed over

the physical storage medium, taking into account the physical constraints of the medium (e.g., track size, tape blocks) with a view to obtaining efficient usage. Normally a user wants to position the bit strings corresponding to his records relative to the physical boundaries of a medium (e.g., tape gaps, beginning of tracks). These boundaries of a medium can occur at several levels. For example, in the case of a magnetic disk, the levels from lowest to highest are: blocks on a track, tracks on a cylinder, cylinders in a disk pack, one disk pack from the number available. These boundaries can be organized in a hierarchy, with each level in the hierarchy split up into several lower levels. We will call the unit of storage at the lowest level of a device (e.g., blocks for a disk track) a basic block; and the unit of storage at each of the higher levels a block. Thus, blocks can contain other blocks and/or basic blocks. We call this structuring of the blocks the storage structure.

Once a user has specified the storage structure, he must specify how the blocks are to be encoded. This is done by specifying encoding characteristics such as length of basic blocks, and the labels that are used to indicate the beginning and end of blocks.

Once the storage structure and its encoding have been specified, the user must specify how he wants his records to be positioned relative to the basic blocks in the storage structure. He may specify, for example, whether there is to be a maximum number of records which are to be positioned within a block; whether records must be maintained

whole, or can be split across blocks; whether any record or only certain record types can be positioned first in a block. He must also specify how any pointers contained in the file are to be interpreted in terms of such characteristics as addressing schemes and mode. The rules for positioning records within blocks and the pointer characteristics determine the addresses which are encoded into bit strings and then used as pointers to encode the structure of the file (see Chapter 4). There must be a mechanized process which, using a description of the file structure and of the record structures, and the record positioning rules and addressing schemes, obtains the actual bit strings for the pointers.

This specification determines the bit string representation of the file in its storage structure.

To encode this bit string onto a particular storage medium, the user must specify medium dependent characteristics, such as to which physical levels of the medium the blocks correspond and the actual physical encoding (e.g., tape density).

We can identify five parts in this process:

(1) The specification of the storage structure;

(2) The encoding of the storage structure;

(3) Rules for fitting records into blocks and implementing pointers;

(4) The resulting bit string representation of the file in its storage structure; and

(5) Medium dependent encoding of this bit string.

We will model here the first four parts only. The transformation to the fifth part from the fourth part is quite straightforward, though hardware dependent, and will be discussed separately in Section 5.6.

We model the storage structure by a set of production rules for specifying the structure of the blocks in terms of basic blocks and block names. The encoding rules of the storage structure are modelled directly by giving the characteristics which must be specified to encode basic blocks and block names. The rules for positioning records and implementing pointers are also modelled directly by giving the necessary characteristics.

The model will be presented by first giving the storage structure part, then the encoding characteristics, and finally the rules for positioning records and implementing pointers.

5.2.1 The Conceptual Structure of Storage

The conceptual structure of storage will be described in terms of storage cell, basic block and block name. A definition of storage item based on these concepts is given.

A storage cell is the basic unit written or read by a storage device as a single character. For example, on tape, this is either a 7- or 9-bit column.

Intuitively, a basic block is a string of consecutive storage cells which are separated from other storage cells by physical delimiters (e.g., tape gaps). When several basic blocks are to be processed in the same way (i.e., they have the same length constraints and contain records positioned in similar ways), each of these basic blocks is

assigned the same block name.

Definition 5-1. A storage item is an ordered pair < n, b > where n is a block name and b is a basic block.

For example, the pairs:

< basic block A, tape block 1 >

< basic block A, tape block 2 >

< basic block E, tape block 3 >

are storage items.

Definition 5-2. A structured set of storage items* (sssi) is any set of storage items which are structured according to the following rules:

sssi → < block name, {block} >

block → block, block

block → sssi

block → storage item

---

* Throughout this discussion, it will be useful to note the analogy between the conceptual organization of sssi's and the conceptual organization of groups (Chapter 3). Both sssi's and groups are given a hierarchic organization, so one should expect a strong correspondence between how they are modelled. The correspondence here is as follows:

block name ≡ attribute
basic block ≡ value
block ≡ compound value
storage item ≡ data item
sssi ≡ group.

For example, the following sssi represents a magnetic tape containing n physical blocks of the same kind:

< tape file X, {< basic block A, tape block 1 >,
< basic block A, tape block 2 >,
...
< basic block A, tape block n > } >

This sssi represents the physically formatted tape illustrated in Figure 5-1.

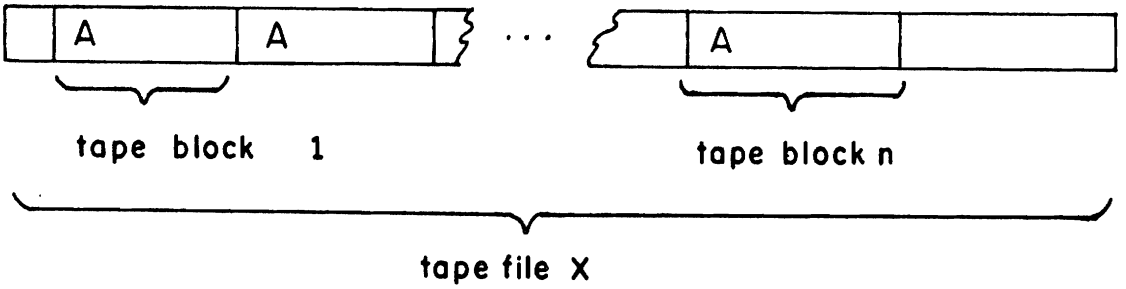


Figure 5-1. Formatted Tape

We now abstract a notion of storage structure for sets of storage items based solely on block name.

Definition 5-3. A storage structure for a set of storage items is a relationship, over the block names of the storage items, which can be produced by the following block productions:

1. storage structure → structure
2. structure → < sssi block name, {substructure}>
3. substructure → substructure, substructure
4. substructure → structure
5. substructure → storage item block name

6. substructure → null

For example, the block names block I, block J, block K, and block L may be related by structures obtained from the following block productions:

storage structure → disk structure

disk structure → < disk file Y, {substructure Y1}>

substructure Y1 → substructure Y1, substructure Y1

substructure Y1 → cylinder structure

cylinder structure → < cylinder, {ssY1Y1} >

ssY1Y1 → ssY11, ssY1YYY2

ssY1YYY2 → ssY12, ssY1YY2

ssY1YY2 → ssY12, ssY1Y2

ssY1Y2 → ssY12, ssY13

ssY13 → ssY13, ssY13

ssY11 → track structure Y11

ssY12 → track structure Y12

ssY13 → track structure Y13

track structure Y11 → < track A, {ssY11Y1} >

track structure Y12 → < track B, {ssY12Y1} >

track structure Y13 → < track C, {ssY13Y1} >

ssY11Y1 → ssY11Y1, ssY11Y1

ssY11Y1 → block I

ssY12Y1 → block J

ssY13Y1 → ssY13Y1, ssY13Y1

ssY13Y1 → block K

ssY13Y1 → block L

One particular structure of block names is:

< disk file Y,

{ < cylinder,

{ < track A, {block I, block I, ... , block I} >,

< track B, {block J} >,

< track B, {block J} >,

< track B, {block J} >,

< track C, {block K, block K, ... , block L} >,

< track C, {block L, block K, ... , block L} >,

< track C, {block L, block L, ... , block K} >,

< track C, {block K, block L, ... , block K} >,

< track C, {block L, block K, ... , block L} >,

< track C, {block L, block K, ... , block L} >}>}>

Replacing block names block I, block J, block K, and block L with storage items, we obtain the sssi for a disk file illustrated in Figure 5-2.

This completes the discussion of the modelling of the conceptual structure of storage. We will now discuss the storage item and storage structure encoding.

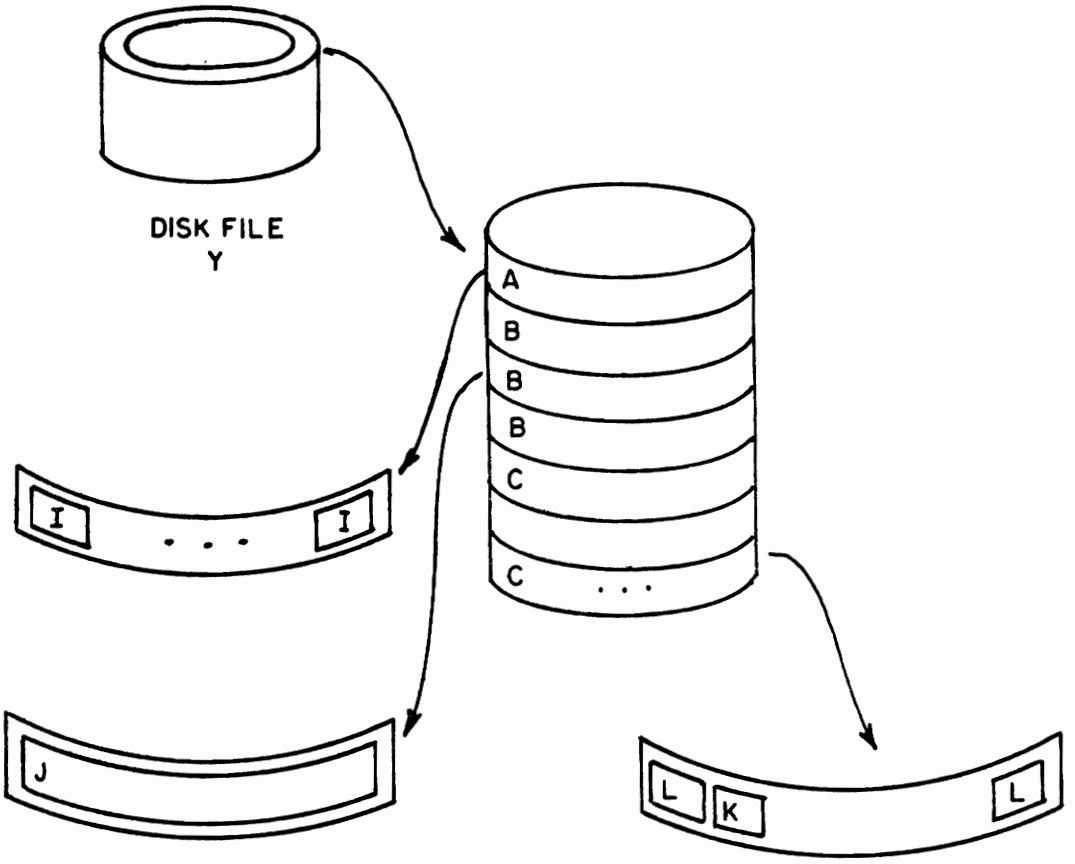


Figure 5-2. SSSI for Disk File

5.2.2 Encoding Storage Items and Storage Structure

This section is analogous to those sections in Chapter 3 on the encoding of values, attributes and structure. We will therefore abbreviate this section by simply listing the characteristics which have to be specified. Storage structure is encoded by encoding block names, in the same way that record structure is encoded by encoding attributes. The following table, Table 5-1, indicates the characteristics required to encode basic blocks and block names.

	1	2	3	4	5	6	7
basic block	X	X					
block name			X	X	X	X	X

Table 5-1. Characteristics required for Encoding

1. Length. The length of a basic block is the number of unit storage cells that it contains (e.g., a column of 7 or 9 bits on a tape).

2. Length Uniformity. If the basic blocks corresponding to a particular block name are always of uniform length, then the length can be described simply by giving the number directly. However, if the length of the basic blocks corresponding to a particular block name are not uniform, then the length is specified as varying.

3. Labels. Labels at the beginning and end of a basic block or block provide one way to encode the block name. Labels can be described simply as a character or bit string.

4. Order. The order of block names can be specified by listing them in the appropriate order. This order can be used to identify the block name of the basic block or block being processed. This provides another way to encode block names.

5. Occurrence. The occurrence of particular kinds of blocks or basic blocks may be mandatory or optional within a substructure.

6. Repetition number. The repetition number is the number of times a block name may occur consecutively in a substructure.

7. Repetition uniformity. If the number of times a block name repeats is always the same (i.e., the repetition of the block name is uniform), then the repetition number can be specified simply by giving the number directly. However, if the repetition of the block name is not uniform, then either the repetition number must be encoded and stored in a label, or the storage items or sssi's containing the block name must be delimited by labels.

This now completes the characteristics which must be specified to determine the rules for encoding storage items and storage structure.

As for the encoding characteristics for record and file descriptions, we allow each characteristic to be specified either:

1) directly - by specifying explicitly the characteristic, or
2) indirectly - by specifying a function which must be computed to determine the characteristic. The function may be defined over the values of data items or other characteristics using the usual arithmetic operators.

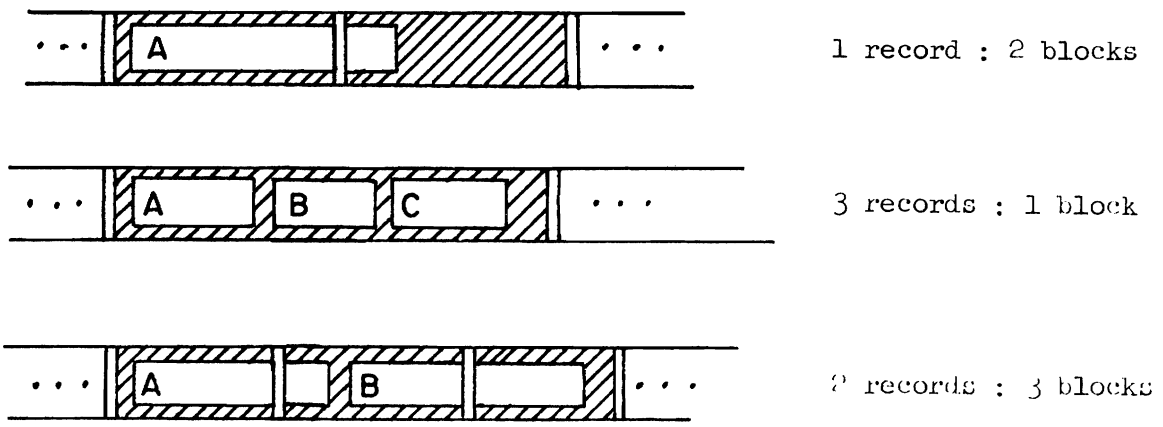
For example, the length of a basic block can be specified directly, or it can be specified indirectly as, perhaps, being equal to the length of another kind of basic block.

5.2.3 Record Positioning and Pointer Interpretation Rules

We have now seen how storage is structured and encoded. The storage structure is the framework within which the user distributes the bit string representation of his file. The user must specify characteristics which determine how his records are positioned relative to the basic blocks, and how the storage addresses of the records are to be determined to produce the pointers for his file.

The user specifies how his records are to be positioned relative to the basic blocks in terms of characteristics which apply to all records (e.g., the number of records per block) and characteristics which apply to particular record types (e.g., whether or not a record of a given type can be split across a block boundary; whether or not a record of a given type can occur first in a block). It is understood that a mechanized process, using the descriptions of the file and record structure, is available which applies these rules to distribute the bit string representations of the records appropriately. The characteristics which specify record positioning are:

1. Record distribution ratio. The number of records stored in basic blocks can be described as a ratio of records per basic blocks. Three examples of such ratios are illustrated below:



2. Record split set. The record split set is the set of those record types whose bit string representations may be split between basic blocks.

When the space remaining in a basic block is not enough to contain the complete bit string of a record type which may not be split, then that bit string must be put into the next basic block. The remaining space in the first basic block is considered to be arbitrarily filled. The contents of such unused space is called filler.

3. Start record set. The start record set is the set of those record types that may occur first in a basic block. For example, if the bit strings representing records of two types, X and Y, are to be positioned in basic blocks such that only records of type X may occur as the first record in a basic block, then the start record set is {X}.

4. Alignment set. The alignment set gives, for each type of record, the set of fields and groups which must be aligned with respect to storage cell boundaries, whether the alignment is to the left or to the right, and the padding characters.

The user specifies how pointers in his file are to be interpreted in terms of the following characteristics:

5. Pointer type. Pointers may give the main memory addresses or device addresses.

6. Pointer mode. Pointers may be interpreted as giving the absolute address of a record, or the address relative to some fixed origin (for example, the first record in the file or a particular block in the storage structure).

7. Addressing scheme. The addresses of blocks may be represented by numbers in ascending or descending order beginning with a particular number. All blocks of the same kind may be numbered sequentially or all blocks within a particular substructure may be numbered sequentially. This characteristic must be specified when pointers give device addresses.

8. Pointer form. When the pointers give device addresses the levels of block addressing used to form each pointer must be specified. For example, a pointer for a disk may consist of just a cylinder number or of a cylinder number and a track number.

This now completes the characteristics which must be specified to encode a file in its storage structure.

5.3 An Application of the Model of Storage Structures

We will now illustrate how the model is used to encode the storage structure of a file into its bit string representation. We will take the file of sequentially ordered person records given in Example 1 of Chapter 4, and position these records relative to the basic blocks in the sssi - tape file X of the above example.

Applying the following encoding characteristics to the storage structure, we obtain the bit string representation illustrated in Figure 5-3.

1. The length of basic block A is 200 bytes (9 bit columns).
2. The length is uniform.
3. There are to be three identical labels, in this case tapemarks: one at the beginning of tape file X, and two at the end of tape file X.
4. Since there is only one kind of block in tape file X, no order is specified.
5. An occurrence of basic block A in tape file X is mandatory.
6. There may be any number of occurrences of basic block A in tape file X.
7. The number of occurrences of basic block A is not uniform among occurrences of tape file X.

The record positioning rules are:

1. The bit string may be split only between records.
2. The distribution ratio is - 1 record : 1 block.
3. The start record set for basic block A is {person}.

4. The alignment set is empty.

Pointers were not used to implement the file structure so no pointer characteristics are specified.

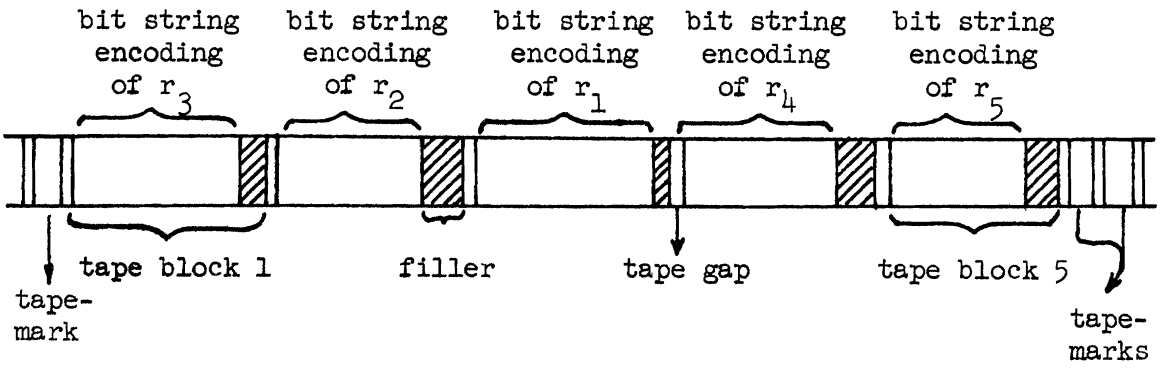


Figure 5-3. Bit String Representation of Tape File X

5.4 The Completeness and Generality of the Model

We will now show that the model of storage structures is complete in the sense that it incorporates all of the storage level characteristics derived in Table 2-1. We will do this by showing that the structuring characteristics at the storage level of Table 2-1 are properly contained in the conceptual part of the model and that the implementation characteristics are contained in the encoding rules of the model. The structuring characteristics for devices supported by software systems allow the user varying degrees of freedom in specifying storage structure at the different device levels. The storage structure permits the description of the organization of such devices at every level. The implementation characteristics and record positioning and pointer interpretation rules are specified directly by the encoding characteris-

tics of the model. Thus, the model is complete in the above sense.

The storage level characteristics of Table 2-1 are incorporated in more generalized forms in the model to allow for the description of variations on existing data structures. This generality is provided in the following ways:

1) The record positioning and pointer interpretation rules provide greater user control than is provided by current software systems.

2) The block productions permit the description of blocking at every device level, instead of adhering to the restrictions imposed by current systems.

3) The model provides a more generalized way to specify characteristics. The encoding characteristics for storage description as well as those for file and record descriptions can be specified as depending on data items, other characteristics and functions of these. This greatly increases the variety of encodings which can be specified.

4) The model can be used to describe storage structures on any device that relies on the basic concepts of blocking and labelling.

In these ways, the model allows variations of current data structures at the storage level to be described.

5.5 The Relationship Between the Model and GDDL

GDDL has been explicitly designed in terms of the model. The BBLOCK and BLOCK statements are used to describe the conceptual storage structure of the model. Each encoding characteristic of the basic blocks and block names can be specified by one or more parameters in GDDL statements. The parameters and statements for these characteris-

tics are listed in Table 5-2 given below:

Basic Block Encoding Characteristics	Statements and Parameters	Remarks	Section in Appendix A
Length	BBLOCK Statement parameter (ii)	Storage Item Characteristics	3.1
Length Uniformity	BBLOCK Statement parameter (iii)	Storage Item Characteristics	3.1
Block Name Encoding Characteristics	Statements and Parameters	Storage Item Characteristics	
Labels	BBLOCK statement parameters (ix) and (x) CONSTANT Statement BLOCK Statement parameter (iv) parameter (v)	Storage Item Characteristics SSSI Characteristics	3.1 1.4.2 3.2
Order	BLOCK Statement parameter (iii)	SSSI Characteristics	3.2
Occurrence	BLOCK Statement parameter (iii)b	SSSI Characteristics	3.2
Repetition Number	BLOCK Statement parameter (iii)c	SSSI Characteristics	3.2
Repetition Uniformity	BLOCK Statement parameter (iii)d	SSSI Characteristics	3.2
Record Positioning Characteristics	**Statements and Parameters**	**Remarks**	**Section in Appendix A**
Record split set	BBLOCK Statement parameter (vii)		3.2
Record distribution ratio	BBLOCK Statement parameter (iv), (v) and (vi)		3.2

Record Positioning Characteristics (cont)	Statements and Parameters	Remarks	Section in Appendix A
Start record set	BBLOCK Statement parameter (viii)	Mapping Characteristics	3.2
Alignment set	FIELD Statement parameter (vii)		1.1
Pointer Characteristics	Statements and Parameters		
Pointer type	POINTER statement parameter (ii)		3.3
Pointer mode	POINTER statement parameter (iii)		3.3
Addressing scheme	BLOCK statement parameter (ii)		3.2
Pointer form	POINTER statement parameter (iv)		3.3
Specification of Characteristics	Statements and Parameters		
Direct	By listed parameters		
Indirect	Parameter and PARAMPROG Statements		1.4.4

Table 5-2 The Relationship Between the Model and GDDL

The way in which BBLOCK and BLOCK statements are used to describe storage structures may be seen by comparing the format of these statements (see Section 3.1 and 3.2 of Appendix A) with the definitions of storage item and storage structure (Definitions 5-1 and 5-3).

The BBLOCK statement has the format:

BBLOCK (block name, encoding and record positioning characteristics)

This corresponds to the definition of storage item.

The BLOCK statement has the format:

BLOCK (block name, ... ; (list), ... , (list), ...)

This corresponds to the block production rules, with (list), ... , (list) corresponding to all the substructures that can be contained in the given structure. Each list may refer to either another block or to a basic block.

Since GDDL is based on the model, GDDL is complete and general in the same way as the model.

5.6 Medium Dependent Encoding Characteristics

So far in this chapter we have shown via the model that GDDL can describe storage structures and their encodings. Now we must consider how such storage structures are tied down to particular media. We must show how the storage structure is related to the various physical levels of a storage medium.

The definition of basic block in Section 5.2.1 requires that a basic block always correspond to the lowest specifiable level of a medium. However, a block may correspond to one of several physical levels of a medium. In a disk pack for example, a block may correspond to a track, several tracks, a cylinder, several cylinders, or a pack. Therefore, each block must be tied down to a particular physical level.

This is achieved by specifying, for each level of a medium, the names of the blocks which correspond to that level. In GDDL this specification is called a device statement (see Appendix A, Section 3.4). A device statement also specifies other medium dependent characteristics such as tape density for tapes. As different media have different numbers of levels and other characteristics, a device statement is needed for each storage medium.

For example, the DISK statement required for the above disk example is:

```
DISK ( ... , pack: 'disk file Y'; cylinder: 'cylinder';
       track: 'track A', 'track B', 'track C' )
```

The device statements then supply all the medium dependent information that is necessary to physically encode files onto storage media.

5.7 Demonstrations of GDDL's Completeness

In the previous section we showed that GDDL is complete for storage description by showing that the model on which it is based is complete. We now provide several practical examples of its completeness. These examples demonstrate the use of GDDL in describing real-world files. The storage descriptions are part of larger examples of complete conversions of data from one structure to another. These examples are given in Appendix B.

CHAPTER 6 DATA CONVERSION

6.1 Introduction

In previous chapters we have seen how to model a file, which is encoded as a bit string on a storage medium, and how such a file may be explicitly described. Thus, the first two objectives of this dissertation have been achieved. Now we will show how such descriptions can be used when data in one structure is to be converted into another structure. This is our third and final objective.

In the first section below, the conversion process is discussed in general terms, and it is shown that additional information in the form of an association list is required to describe data conversion. Section 6.3 gives a model of the concept of an association list which satisfies the requirements discussed in the previous section. Section 6.4 illustrates the model by giving some applications of it in examples of data conversion. Section 6.5 shows how GDDL's ability to describe this aspect of data conversion is based directly on the model of the association list. Finally, Section 6.6 ties all the previous work together by showing how and where each part of the necessary descriptions is used during the conversion of data from one structure to another.

6.2 The Concept of the Association List

The present discussion apparently marks the first time that anyone has considered precisely what information must be made explicit to

use data descriptions for data conversion. Therefore, we must develop our ideas from basic concepts, and so a longer intuitive discussion will be needed. This intuitive discussion will be organized as follows. First we will specify exactly what data conversion is, using terms developed in previous chapters. Then we will take a rather simplified view of data conversion to introduce our basic notion - that of an association list. Finally, we will show how this notion of an association list needs to be made more elaborate so that all the requirements for specifying data conversion are met. We begin then by providing a definition of data conversion.

Definition 6-1. Data Conversion is a process which, given the bit string representation of a file (or set of files) on a storage medium, produces the bit string representation on a (different) storage medium of a file (or set of files) whose data items contain values from the data items in the first file (or set of files). The structure of the first file(s) is different in general from the structure of the second file(s).

At this point, it will be convenient to introduce some new terminology.

Given a set of files $X_1, \ldots, X_n$ whose data are to be converted into a set of files $Y_1, \ldots, Y_m$; we call the files $X_1, \ldots, X_n$ the source files and the files $Y_1, \ldots, Y_m$ the target files.

Note that from the above definition, the value of a target data item must always be the value of some source data item. As the data

item is the most primitive structure in the model, we do not provide the capability for decomposing a source value into smaller components and using these components for target values. For example, if "TOMJONES" is a source value, the only target value that may be obtained from this value is also "TOMJONES" and not just "TOM" or "JONES". This is not a serious restriction as a user would normally form a separate data item for each type of value to be processed independently. Thus, in the above case, "TOM" might be the value of one data item and "JONES" the value of another data item, and both data items would be structured as a source group.

Note also that from the above definition, data conversion is not necessarily a reversible process. That is, if file A is converted into file B, it does not follow that file B can be converted back to file A. A simple counter-example is when file B only contains a subset of the values of data items in file A.

We will now develop the basic idea of the process of data conversion. Consider the case of a user who wishes to convert data from a single source file to a single target file. We will assume that the user has the bit string representation of the source file stored on a storage medium, and that he has descriptions of the source and target files. The object is to form the bit string representation of the target file.

This object can be achieved in essentially the following way:

1) Use the description of the source file to break down the bit string representation so that the actual data items (attribute-value pairs) are obtained.

2) Provide a specification of how the values of source data item attributes are to be combined with target data item attributes for the target file.

3) Form the target data items according to this specification.

4) Structure and encode these data items according to the description of the target file to obtain the bit string representation.

We see that the only additional information that the user needs to provide is a list (corresponding to item 2 above) which gives, for each target attribute, the source attribute which is to provide its value. We will call such a list an association list because it associates each target attribute with a source attribute. Thus, an association list is a set of target attribute-source attribute pairs.

The essence of this conversion process is illustrated in Figure 6-1. Emphasis is placed here on the role of the association list, because it is this concept that we now want to develop. We will wait until Section 6.6 to give a detailed treatment of how the various parts of the source and target descriptions relate to the conversion process.

With this concept in mind, we can go on to a more elaborate discussion of the requirements for an association list to completely describe associations between source and target attributes.

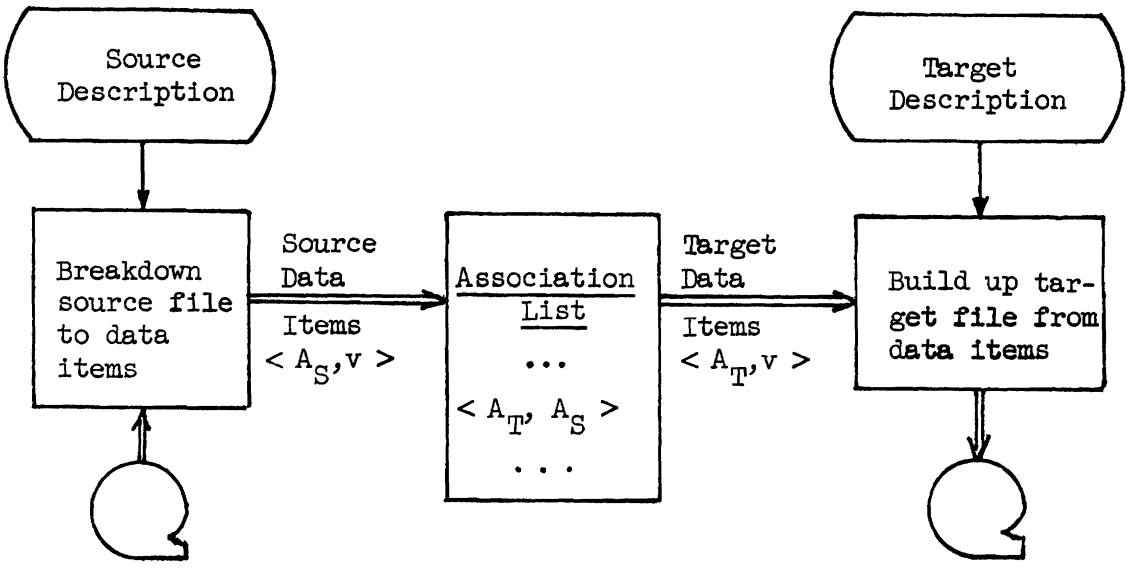


Figure 6-1. Simplified Conversion Process

The above view of an association list, as a set of target attribute-source attribute pairs, is adequate only if all the data items of a target record are obtained from a single source record. However, in general, the value for a data item of a target record may be obtained from any data item in any record in the source file. There are two cases to consider. First we consider when the data items of a given target record are obtained from several different source records. In this case we must identify which source record contains the appropriate value of the source attribute.

Secondly, consideration must be given to repeating groups and fields within the source record. It may be necessary to identify from which occurrence of the repeating group or field the value of the source attribute is to be taken. If it is known that the jth occurrence is to provide the value, this can be specified by indexing the source attribute appropriately in the association list. Otherwise, we must develop a further means for identifying appropriate occurrences.

We can summarize the requirements as follows. To properly identify the source attribute for a target attribute - source attribute pair, the association list must include the capability of identifying:

1) the required source record, and
2) the required occurrence of a repeating group or field in the source record.

We will first consider how the required record can be identified. The problem here is similar to the one in Chapter 4 for specifying which records are connected by access paths. In that chapter the solution was to provide criteria which determine that two records are to be linked if they satisfy the criteria. We can use a similar solution here. We can specify criteria which select a record to supply a value for a source attribute if that record satisfies the criteria. In this way we can use the power of the criterion production system (cps) which we already have available, and avoid constructing additional mechanisms.

We saw in Chapter 4 that the cps is a general system for producing any criteria over records in terms of values, characteristics and paths.

However, the records which could be named (see record-modifier in cps) in these criteria were only head record, tail record, and arbitrary (variable) records. To use this criterion production system for present purposes, we shall see that three extra productions must be added to provide new ways of naming records which are involved in criteria. In the following discussion we will determine what source records need to be named, and develop a way to name them appropriately. It will turn out that there are occasions when the names we develop can be used by themselves to identify directly appropriate records in the association list.

We first observe that the data in the source file is organized in some way which is meaningful to the user. This is very apparent from the models of record and file description in previous chapters. Similarly, the target file will be another meaningful organization of this same data. Consider the implications of this observation for the criteria that the user will want to introduce into the association list to identify particular records. Let us assume the user needs to write criteria for some target record that is to contain attributes A_{T_1} and A_{T_2}. He has decided that the source attributes which are to provide the values will be A_{S_1} and A_{S_2}. The association list will therefore contain the pairs $< A_{T_1}, A_{S_1} >$ and $< A_{T_2}, A_{S_2} >$. Let us assume further that the record containing the occurrence of A_{S_1} which provides the value for A_{T_1} has been selected. Now the user wants to specify criteria for selecting the record containing the occurrence of A_{S_2} which provides the value for A_{T_2}. Because the data items in the target record are supposed to be related to each other, he may want to use the values of some data

items in the source record, which provides the value for A_{T_1}, in selecting the record containing the occurrence of A_{S_2}.

For example, assume each source record contains an attribute A_{S_i}. The user may want the record which contains the occurrence of A_{S_2} (providing the value for A_{T_2}) to have the same value for A_{S_i} as the value for A_{S_i} in the record which provides the value for A_{T_1}. This is illustrated in Figure 6-2.

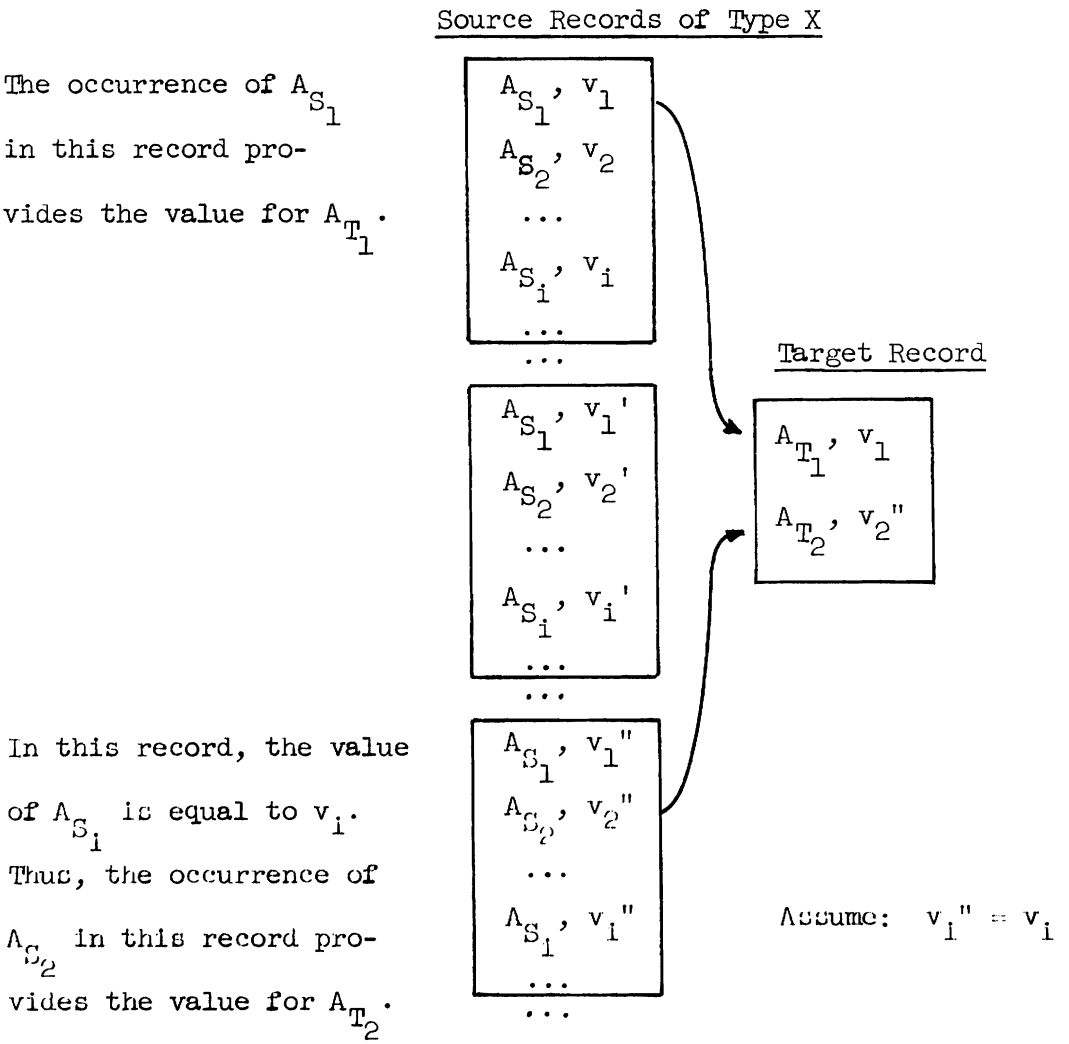


Figure 6-2. An Example of Source Record Selection for the Formation of Target Records

The above example illustrates the case when the target record contains two attributes. In the general case when the target record contains n attributes, A_{T_1}, A_{T_2}, ... , A_{T_i}, ... , A_{T_n}, the selection of the source record which provides the value for A_{T_i} $(2 \leq i \leq n)$ may depend on values in those source records which provide the values for A_{T_1}, ... , $A_{T_{i-1}}$. We will see an example of such a case in Section 6.4.

We must, therefore, provide the criterion production system (cps) with the capability of referring to a record that has already provided a value for some target attribute. Let us introduce the terminology SOURCE (A_T) to mean the source record which provides the value for A_T. Now we can say the selection of the record providing the value for A_{T_i} may depend in general on the values of data items in SOURCE (A_{T_1}), ... , SOURCE $(A_{T_{i-1}})$. Using this terminology we can express the criterion in the Example of Figure 6-2 for selecting the source record containing the occurrence of A_{S_2} which provides the value for A_{T_2} as: A_{S_i} OF X = A_{S_i} OF SOURCE (A_{T_1}).

We have thus determined a way of naming a source record relative to the target attribute for which it contributes the value. This way of naming source records provides a method for obtaining criteria to identify appropriate source records in the association list. In addition, if the record which contains a source attribute is required to be SOURCE (A_T), for some A_T, we can use this name itself for identifying the appropriate record. This is the case in the association list provided for Example 1 in Section 6.4. These capabilities satisfy the first requirement above.

We now consider the second requirement when the value for a target attribute is to come from a source attribute in a repeating group or field (see Section 3.3). The naming problem here is analogous to the above case when a value for a target attribute comes from a particular source record. That is, the user may want subsequent values for other target attributes either to come from the same group or to be determined by other values in that group, or to be determined by encoding characteristics of that group or field. Thus, we need a way of referring to source groups or fields that have already been used to supply values to the target record. We can name them by appending the attribute of the group or field in question to our previous term SOURCE (A_T). For example, to refer to a repeating group of type X which was the source of the value for target attribute A_T, we say X-SOURCE (A_T). If we wish to refer only to a higher level group of type Y which contains X, we say Y-SOURCE (A_T). We have thus determined a way to name a repeating source group or field relative to the target attribute for which it contributes the value. Again this name may either be used as part of criteria which identify the required occurrences of the repeating group or field, or the name may be used by itself when it happens to name the required occurrence. These capabilities satisfy the second requirement above.

There is one additional situation to consider that is related to the two requirements above. This situation occurs when a <u>target</u> attribute happens to be the attribute of a repeating target field or group. We may have to identify the particular occurrence of a target attribute

A_T when it appears in names of the form SOURCE (A_T). To do this we must make our naming method a little more elaborate. If the number of the occurrence is known, then the attribute A_T can simply be indexed. However, if the number of the occurrence is not known we must make provision for a criterion which identifies A_T. Thus, to refer to the source of the value for a target attribute A_T in a repeating (target) group or field, we use either the name SOURCE (A_T index) or the name SOURCE (A_T, criterion).

We have thus seen how to name records, and thence groups and fields which may be used in defining criteria to select particular occurrences of source attributes. This naming method together with the cps of Chapter 4 allows criteria to be specified which meet the two requirements for properly identifying source attributes appearing in the association list.

So far we have only considered conversion of a single source file to a single target file. In the general case of converting several source files to several target files, the association list must specify for each target attribute-source attribute pair, the file to which the target attribute and the file to which the source attribute belong.

In the next section we will present a model of the concept of association list based on the discussion above. We will then give some examples of the application of this model in data conversion.

6.3 A Model of the Association List

Definition 6-2. An association list is a set of six-tuples of the form:

< target attribute, target file name; source attribute, source file name; record identification, attribute repetition identification >

where: 1) target attribute is the attribute in the target record to be provided with a value,

2) target file name is the name of the target file in which the target attribute occurs,

3) source attribute is the attribute in the source record which is to provide the value for the target attribute,

4) source file name is the name of the source file in which the source attribute occurs,

5) record identification is optional; it is either a name for a record or a criterion which can be expressed using the criterion production system specified below, and which is used to identify the source record in which the source attribute occurs,

6) attribute repetition identification is optional; it is either a name for a group or field or a criterion which is used, when the source attribute occurs in a repeating group or field, to identify the particular occurrence in which the source attribute occurs. When the particular repetition is always uniform and

mandatory, the source attribute is simply indexed; otherwise, the criterion is expressed using the criterion production system specified below.

Criterion Production System:

This system contains the productions of the system in Chapter 4, and, in addition, the productions:

source-reference → SOURCE (attribute-modifier)
→ SOURCE (attribute-modifier, criterion)

record-modifier → source-reference

attribute-modifier → attribute – source-reference

These productions allow a record, group or field to be named as that record, group or field in the source file which provides the values for the given attribute in the target file.

The following convention is to be observed in specifying an association list: When a source attribute A_S repeats and

1) when a target attribute A_T does not repeat, then specifying
   a) $< A_T; A_S; \ldots >$ implies that one target record is to be formed for each value of A_S (i.e., if there are n such values of A_S, then n target records are formed);
   b) $< A_T; A_S(i); \ldots >$ implies that only one target record is to be formed and the remaining values of A_S are to be discarded (i.e., are not to be used as values for A_T in other target records);
2) when a target attribute A_T repeats an unlimited number of times, then specifying $< A_T; A_S; \ldots >$ implies that A_T will repeat exactly as many times as A_S repeats;

3) when a target attribute A_T repeats either a fixed or bounded number of times, say m, then specifying

a) $< A_T; A_S; \ldots >$ implies that whenever the number of A_T repetitions is less than the number of A_S repetitions, then target records are to be formed such that each value of A_S appears in some target record;

b) $< A_T(1); A_S(1); \ldots >$

$\ldots$

$< A_T(m); A_S(m); \ldots >$ implies that whenever the number of A_T repetitions is less than the number of A_S repetitions, then only one target record is to be formed with the ith value of A_S as the ith value of A_T and the remaining source values of A_S are to be discarded.

6.4 Applications of the Model of the Association List

Example 1. Extraction of a New File from an Existing File

Consider a source file F1 whose records are described in the following way:

i) The structures of the records are described by the set of productions P1:

record structure → structure R1

structure R1 → < person, {substructure R1R1}>

substructure R1R1 → substructure R11, substructure R1R2

substructure R1R2 → substructure R12, substructure R1R3

substructure R1R3 → substructure R13, substructure R14

substructure R11 → name

substructure R12 → age

substructure R13 → sex

substructure R14 → null

substructure R14 → substructure R14, substructure R14

substructure R14 → structure R14

structure R14 → < book, {substructure R14R1} >

substructure R14R1 → substructure R141, substructure R14R2

substructure R14R2 → substructure R142, substructure R143

substructure R141 → title

substructure R142 → pages

substructure R143 → date

ii) The encoding of the records is specified by a set of characteristics C1 (the exact specification of these characteristics is not required for the purpose of this example).

Consider a target file F2 whose records are described in the following way:

i) The structures of the records are described by the set of productions P2:

record structure → structure R2

structure R2 → < person, {substructure R2R1} >

substructure R2R1 → substructure R21, substructure R2R2

substructure R2R2 → substructure R22, substructure R23

substructure R21 → name

substructure R22 → age

substructure R23 → sex

ii) The encoding of the records is specified by a set of characteristics, C2 (omitted here).

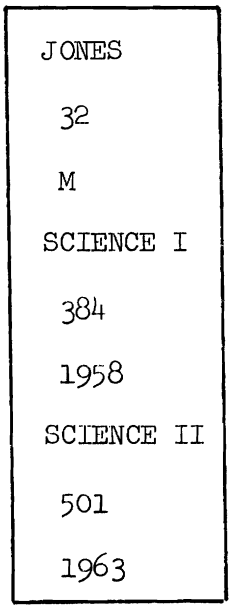
To convert data in source file F1 to the form of target file F2, the following association list is provided:

< name, F2; name, F1 >

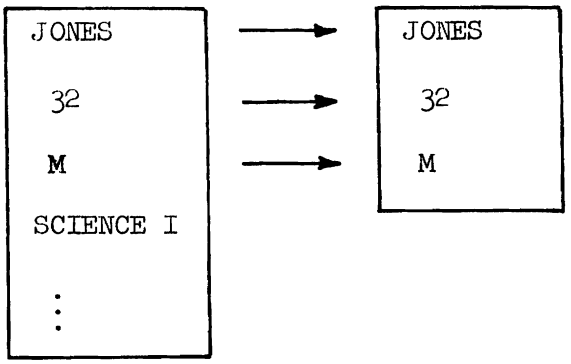
< age, F2; age, F1; SOURCE (name) >

< sex, F2; sex, F1; SOURCE (name) >

Given a record:



it is converted as follows:



All the source records of F1 are converted in this way.

Example 2. Reorganization of a File

Consider the same source file F1, described in Example 1, and a target file F3 whose records are described in the following way:

i) the structures of the records of F3 are described by the set of productions P3:

record structure → structure R3

structure R3 → < book, {substructure R3R1} >

substructure R3R1 → substructure R31, substructure R3R2

substructure R3R2 → substructure R32, substructure R3R3

substructure R3R3 → substructure R33, substructure R34

substructure R31 → title

substructure R32 → substructure R32, substructure R32

substructure R32 → author

substructure R33 → date

substructure R34 → pages

ii) the encoding of the records is specified by a set of characteristics C3 (omitted here).

To convert data in source file F1 to the form of target file F3 such that there is one target record for each value of 'title' in a source record and each target record contains the 'author' from all source records containing the same 'title', the following association list is needed:

< title, F3; title, F1 >

< author, F3; name, F1; criterion >

in a new target record. To find values for the target attribute 'author' all records are checked to see if they contain a value of 'title' equal to the value of 'title' obtained for the target attribute 'title' (in this case SCIENCE II). The two records shown above contain such a value. Therefore, they are used as sources for the values of the target attribute 'author'. In this way, the values JONES and ROE are obtained. Finally, the values for 'date' and 'pages' are obtained from the same group 'book' in the same record which was the source of the value SCIENCE II. In this way, the target record for SCIENCE II is formed.

6.5 The Relationship Between the Model and GDDL

The model of an association list defined in the previous section provides a means for explicitly stating how target data items are formed from source data items during conversion.

GDDL's ability to describe data conversion has been defined in terms of this model and thus provides similar capabilities.

We will now show how the model and GDDL are related. GDDL's ASSOCIATE statement (see Appendix A, Section 2.3.1.1) is an exact image of the association list six-tuples. Target and source file names appear as part of the target and source names (parameters i) and ii)). The SOURCE (attribute-modifier, criterion) naming scheme appears explicitly as GDDL's SOURCE statement (see Appendix A, Section 2.3.1.3).

Thus, we conclude that GDDL can specify any association list that can be defined using the model.

6.6 The Conversion Process

The association list completes the information needed to describe explicitly how data is to be converted from one organization to another. In this section we will see how and where each component of the description for the source and target files together with the association list is used during the conversion process.

In Figure 6-1, we showed that the conversion process consists of essentially three parts. First, the source file is broken down into its component data items using the source description, the target data items are formed using values obtained from source data items, and lastly the target data items are structured and encoded according to the target description. Figure 6-3, which is a detailed treatment of the conversion process, essentially reflects these same three stages in the instance of conversion from several source files to several target files.

Figure 6-3(a) shows how source descriptions are used to read the source files from the storage media and break the bit string representation down into data items, and how the association list controls the process.

Figure 6-3(b) shows how the target data items are formed, and Figure 6-3(c) shows how these data items are organized into a target file and written onto the storage media.

Figure 6-3 is not an algorithm for converting data. It only shows the order in which description components are used for extracting a single data item from a source file, and for converting the value of

this data item into part of the target file. In conversion proper, when large numbers of data items must be extracted, much of the processing for each data item will be done in parallel with that for other data items for efficiency considerations.

Let us follow the conversion process using Figure 6-3.

We will assume that the process is underway and several records for a particular target file have already been constructed. Some of the data items for the next target record have already been formed and we will now follow the formation of the next data item.

The target record structure determines the attribute for this next data item. We must now begin at the top of Figure 6-3(a).

The association list (1) identifies which source file contains the attribute whose value will be combined with the target attribute.

The storage structure description (2) for that source file is used to determine which blocks must be read (i.e., which blocks contain records of the file).

The storage encoding characteristics (3) are needed to read these blocks off the storage medium and to remove any labels.

Once the bit string representation of the file is obtained, the association list (4) identifies which source record is needed. To locate and extract the bit string representation of the record, the criterion used for sequencing the records (5) and the file encoding characteristics (6) are used. If the association list (4) contains a criterion for identifying the source record, many records may have to be extracted and tested against this criterion. If the file contains

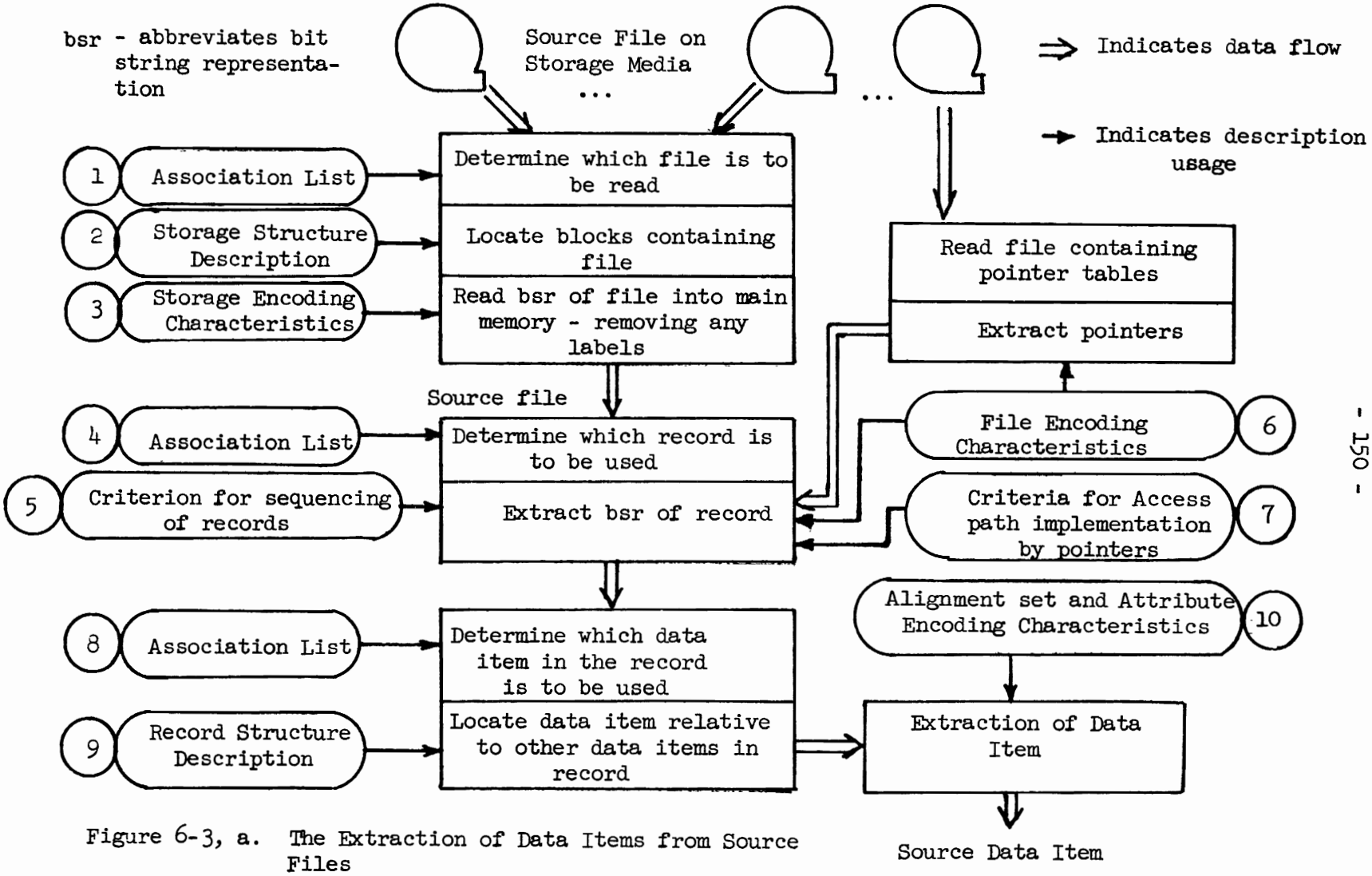


Figure 6-3, a. The Extraction of Data Items from Source Files

Figure 6-3. The Use of Descriptions and the Association List in Data Conversion

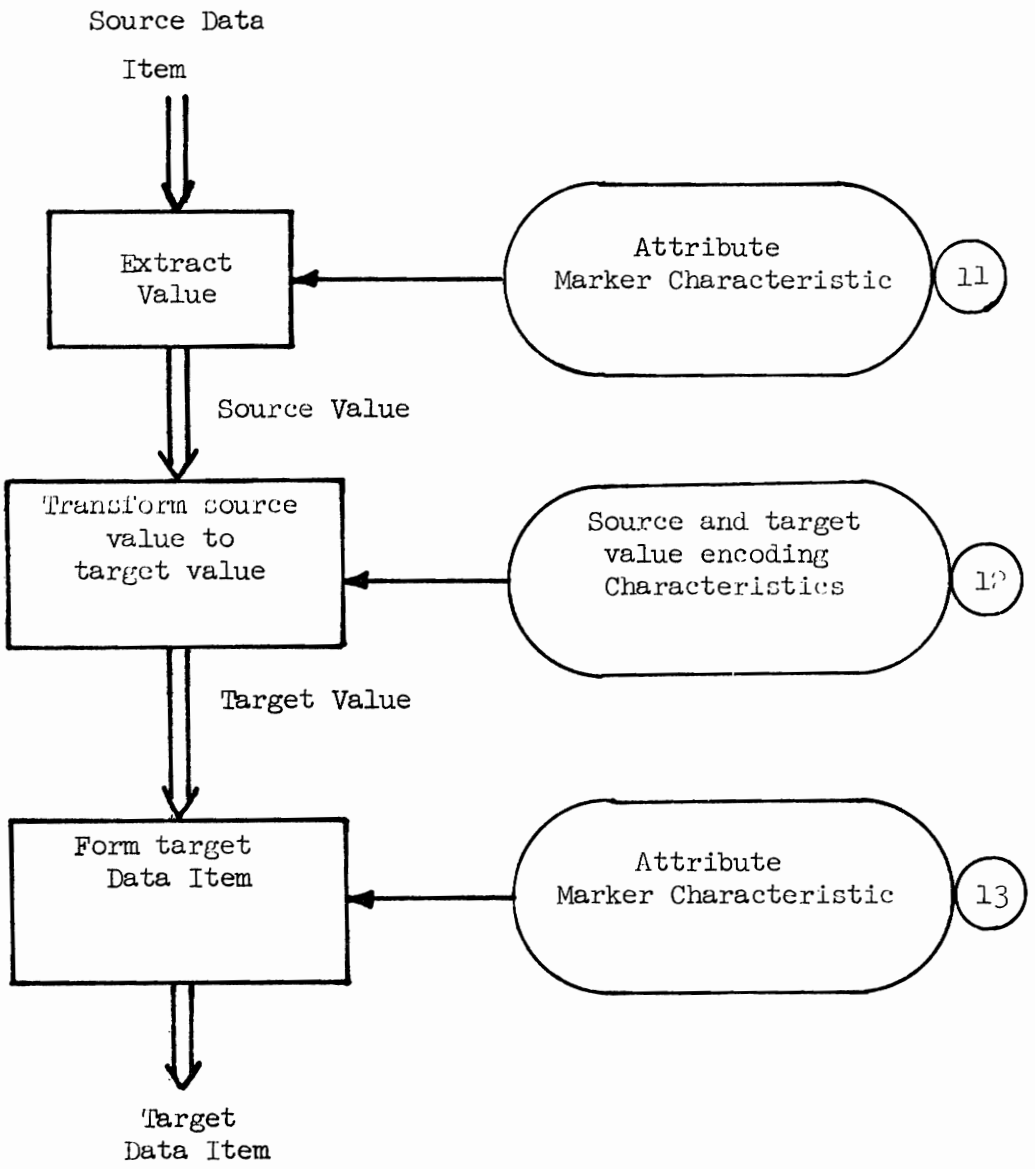


Figure 6-3, b. The Formation of Target Data Items From Source Data Items

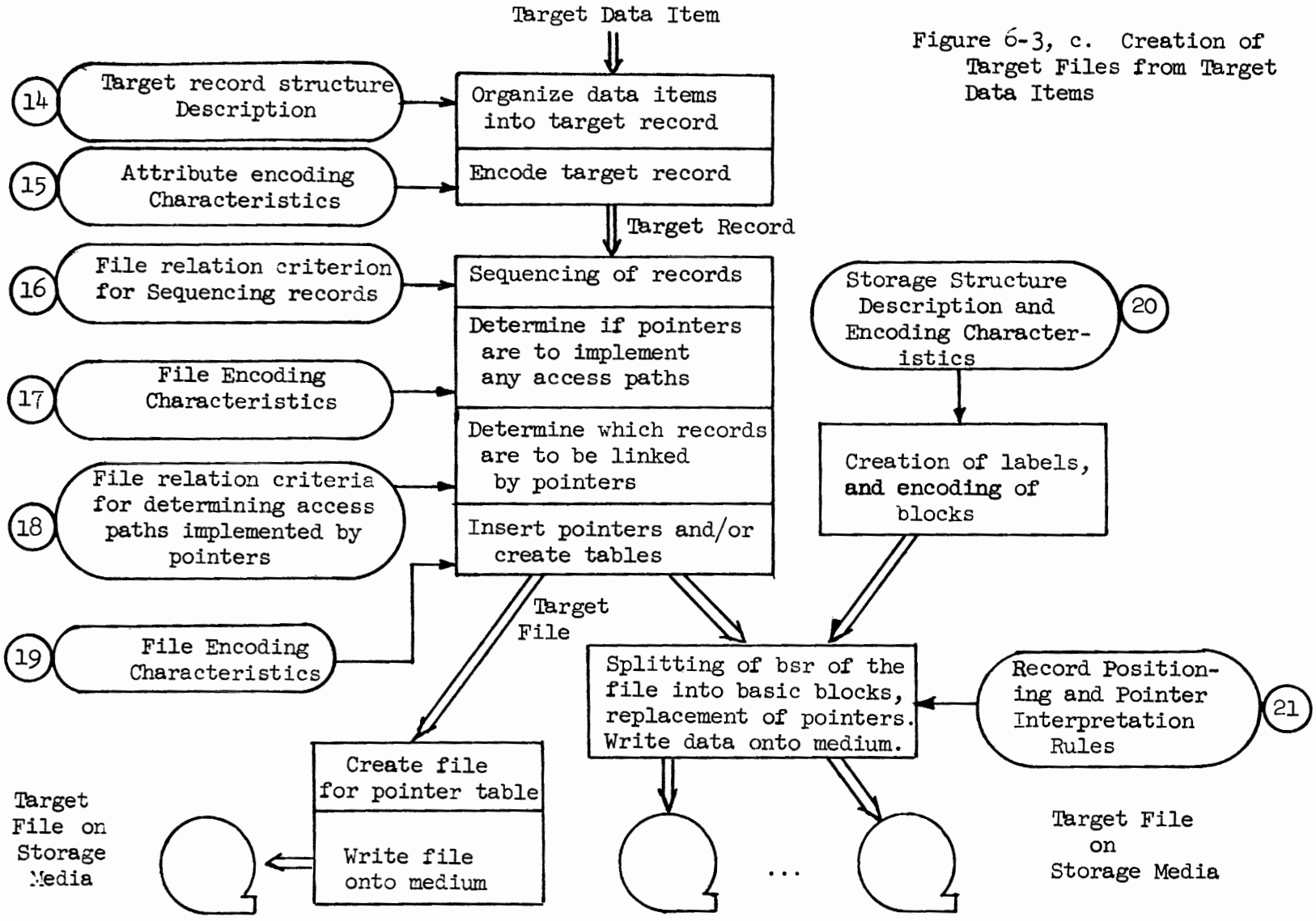


Figure 6-3, c. Creation of Target Files from Target Data Items

access paths which are implemented by pointers, the access path criteria (7) may be used to locate the source record. When the pointers are stored in tables, the appropriate file containing the table must be read and then the file encoding characteristics (6) must be used in extracting the records containing the pointers.

Once the bit string for the source record has been located and extracted, the association list (8) identifies the attribute of the data item in the record which is to be extracted. The record structure description (9) for the record gives the location of the attribute relative to the other attributes in the record. The attribute encoding characteristics and the alignment set characteristics of the storage description (10) are then used to extract the data item from the record. The alignment set characteristic is needed to recognize fields which are preceded by pad characters aligning them with respect to storage cell boundaries.

This completes the process of extracting the desired data item from the source file.

The process of forming the target data item from the source data item just extracted is illustrated in Figure 6-3(b).

The target value is obtained by removing attribute markers as specified by the attribute marker characteristic (11) . Then the value encoding characteristics of the source and target descriptions (12) are compared to determine any changes that must be made in the value string (e.g., adding or removing pad characters to create the correct length, converting from one character code to another). Target attri-

bute markers as specified by the attribute marker characteristic (13) are concatenated onto the value string and it is ready to be used as a target data item in generating a target record.

We will now follow the creation of a target file to explain the processes illustrated in Figure 6-3(c).

The target data items which have been obtained by the process above are organized according to the target record structure description (14) . Then the structure is encoded using the attribute encoding characteristics (15) .

As the target records are generated they are sequenced according to the criterion (16) which determines sequencing. The file encoding characteristics (17) for the target file are checked to see if tables of pointers or embedded pointers are to be created. If this is the case, the criteria (18) are used to determine which records are to be linked by the pointers. The encoding characteristics (19) are used to create the pointers. When the pointers are to be stored in tables, the creation of the tables is treated as the creation of another file.

While the target records are being generated, sequenced, and the required access paths encoded, the storage structure description (20) is used to set up the storage format on the devices indicated. Finally, the record positioning and pointer interpretation rules (21) are used to break the bit string representation of the file into blocks for insertion into the storage structure. At this time, fields are aligned and actual addresses for the media in question are used to implement

pointers. The data is then read onto the medium and the target file is created.

CHAPTER 7 CONCLUDING REMARKS

We conclude this report by assessing the contributions and implications of this work for the data processing field.

The work on data description presented in this report is just the beginning of a new important area. In many ways, the development of data description languages is analogous to the development of higher level programming languages. Programming languages were developed to make it easier for humans to prescribe algorithms to machines and to communicate algorithms among themselves. Data description languages are needed to make it easier for humans to describe data structures to machines and to communicate data structures among themselves. Programming languages have vastly increased the power and applicability of computers, and it is anticipated that ddl's will equally stimulate the data processing field.

The development of programming languages was the result of many studies of their theoretical and practical properties, in such areas as linguistics, automata theory, compiler design, and parsing algorithms. It is expected that data description languages will require an equally intensive study before their power and applicability are thoroughly understood. It is important that this study proceed on abstract and practical planes simultaneously, so that each may stimulate the other and neither will lose sight of their intended goals.

This report has contributed to the study of data description languages in both these planes. First we have developed an actual language which can be implemented to study the practical problems in the design of a GDDL processor*. This implementation will provide the first real measures of the efficiency and effectiveness of ddl's. Secondly, we have contributed on a more formal plane by providing the first complete models of data structure. Before a theoretical study of data description languages can be undertaken, we must ensure that we sufficiently understand data structures to know what must be formalized. These models provide a first step toward a formal study of ddl's.

In addition, we chose to examine in detail one major application of a ddl - namely, the conversion of data from one structure to another. We found that information about the relationship between the two structures must be provided in addition to the description of the structures themselves. We have developed a method for stating this information based on the concept of the association list. This enabled us then to show how data can be converted using ddl descriptions and where the parts of the descriptions are used in the process.

We will now make some suggestions for future research related to data description languages.

As a general comment, we stress the importance of developing complete models of data structures other than the one presented here.

---

* GDDL is being implemented at the Moore School of Electrical Engineering, University of Pennsylvania (Ra 1971).

This is too new a field to concentrate totally on one single approach. There is much to be gained by developing and implementing ddl's based on radically different concepts.

One suggestion for research is based on the following observation. Traditionally, computer scientists have emphasized a dichotomy in information processing, namely, the algorithm which processes the data, and the data which is processed by the algorithm. This emphasis seems to be largely due to the disproportionate effort devoted to the study of algorithms in comparison to data structures. The work in this report suggests a more fruitful view of information processing might be to regard it as a trichotomy - namely, the algorithm, the data structure for that algorithm, and the access method by which the algorithm obtains data from that data structure.

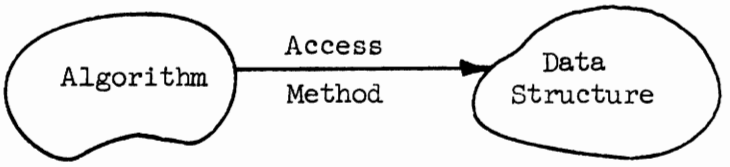


Figure 7-1. The Trichotomy of Information Processing

It has been a central theme of this report that data structures are understood and manipulated best by separating the specification of the structures from the specification of the procedures that access data in the structures. However, programming language theorists, particularly those in extensible languages, tend to relegate everything that is not "purely algorithmic" to the data structure including the

access methods. We, therefore, suggest that access methods should be treated as a separate entity. It is not necessary to associate an access method with a single data structure or vice versa. For example, a list can be accessed as a pushdown or a queue. There are many models of algorithms which have been studied, now we have introduced a model for data structures. Future research should be devoted to studying how algorithms can effectively access (i.e., store and retrieve) data from a data structure. This can be achieved in terms of a model for algorithms and a model for data structures. The specification of such access methods would then become a separate component of a programming language or a part of the languages associated with Generalized Data Base Management Systems.

We will now use an analogy with language theory to suggest a theoretical framework for studying data structures and data conversion.

We have seen how a ddl description specifies the structure of a file, such that if we are given the values for a set of data items, we can encode these data items in accordance with the structure to obtain a bit string representation of the file. Let us define a "file language" to be the (infinite) set of bit string representations of files which could be obtained, depending on the number and values of the data items, for a single ddl file description. We can regard each bit string representation of a file as a sentence in a "file language". Thus, a ddl description can be considered to be the syntax of a file language. The ddl itself is therefore a metalanguage for specifying file languages.

Data conversion then is an exercise in syntax-directed translation, in that we input the syntax of two file languages specified in the ddl metalanguage and a sentence in one of the file languages. The output of the conversion is then a sentence in the other file language. It would be interesting to study file languages as linguistic entities. Language theory may offer insights for either specifying data description languages or producing a new model of data structures based on linguistic concepts.

It is interesting to note that in one sense data conversion is a generalization of natural or programming language translation. In data conversion we need the association list to relate names in one data structure description to names in the other data structure description. The corresponding entity in natural language translation is a dictionary which relates words in one language to words in the other language. Clearly the association list is a more alaborate notion than a dictionary - in fact, just the target attribute-source attribute pairs themselves correspond to such a dictionary. The reason for this elaborateness is that in language translation local structures are preserved (e.g., clauses remain clauses, phrases remain phrases, and sentences remain sentences), whereas, in data conversion local structures need not be preserved in that data items in a target record may be obtained from many source records. The notion of association list is therefore a generalization of a language to language dictionary. This might have in itself some linguistic implications.

Apart from the general ideas outlined above, we suggest the following research possibilities which are directly related to the model and language presented herein.

(1) To implement a GDDL interpreter and data convertor.

(2) To formalize the model more completely by abstracting the implementation characteristics at each level.

(3) To obtain a mathematical or formal characterization of the classes of data structures which can be converted from one to another using the association list developed in Chapter 6.

(4) To enhance the user convenience of GDDL by incorporating such features as extensibility or a macro capability into GDDL.

Appendix A

REFERENCE MANUAL

FOR

GDDL

(Generalized Data Description Language)

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

INDEX A

0. Basic Elements

0.1 The Character Set

The Character Set is composed of digits, alphabetic characters, and special characters. There are ten digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are decimal digits;

0, 1 are binary digits.

There are twenty-six alphabetic characters:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

There are twenty-nine special characters:

Character name	Character
comma	,
semicolon	;
colon	:
period	.
question mark	?
exclamation	!
apostrophe	'
quote	"
open parenthesis	(
close parenthesis	)
space	
number sign	#
ampersand	&
asterisk	*

Character name	Character
at the rate of	@
cents	¢
dollar sign	$
lozenge	◊
underline	_
plus	+
minus	-
slash	/
equals	=
less than	<
greater than	>
percent	%
logical AND	∧
logical OR	|
logical NOT	¬

The characters in the character set are the most primitive elements of GDDL. The characters are combined to form strings called names and strings called numbers.

0.2 Numbers

An integer is a string of digits

Examples: 0
124
03791

A signed integer is the character "+" or the character "-" followed by an integer.

Examples: +1
-124
+03791

A decimal number is an integer or a signed integer, followed by the character ".", followed by an integer.

Examples: 9.012
-102.0
+1378.999

A number is an integer, a signed integer or a decimal number.

0.3 Names

A name is any string of characters from the character set.

An index is an integer enclosed by the characters "(" and ")".

Examples: (10)
(0)
(131)

Note: From the above definition, signed integers and decimal numbers cannot appear in indices.

A user-defined name body is any string of the alphabetic and numeric characters and the characters: space, ., -, and /.

Examples:	GROUP	NAME	FALSE
	-102.0	RCD1	BOOK-GRP

An unindexed user-defined name is a user-defined name body enclosed by apostrophes.

Examples:	'GROUP'	'NAME'	'FALSE'
	'-102.0'	'RCD1'	'BOOK-GRP'

An indexed user-defined name is an unindexed user-defined name followed by an index.

Examples: 'NAME'(10) 'FALSE'(13)
'-102.0'(1)

A user-defined name is either an indexed user-defined name or an unindexed user-defined name.

Examples: 'GROUP' 'NAME'(10)

A system name is any of the names listed in Index A.

0.4 Statements

Statements in GDDL have a fixed format consisting of a system name, followed by a sequence of parameters enclosed in parentheses.

The statements are grouped together into four sections according to their use:

1. Record specification statements,
2. File specification statements,
3. Storage specification statements, and
4. Conversion statements.

Each statement in GDDL is explained in a separate subsection, in which the format of each statement is presented, followed by a description of each parameter. A discussion of the usage of each statement, together with examples, is then given.

Note: In presenting statements formats, the following conventions are used:

(i) square brackets indicate optional parameters.

(ii) "parameter x, ..., parameter x" means that parameter x may occur in the statement one or more times.

(iii) "[parameter x, ..., parameter x]" means that parameter x may occur in the statement zero or more times.

1. Record Specification Statements

The following terminology will be used in presenting the Record Specification statements:

(i) A field is a string of characters or binary digits representing a data item.

(ii) Two fields are of the same type if and only if they are referred to by the same name and are implemented in the same way.

(iii) A group is an organization of fields and/or other groups.

(iv) Two groups are of the same type if and only if they have the same organization and are implemented in the same way.

(v) A record is a group which is to be used as a basic unit of storage and retrieval.

The Record Specification statements are used to specify the organization and implementation of records in terms of groups and fields.

1.1 FIELD Statements

format

FIELD (field name, character code, length type, length, uniformity, data type [; F, alignment factor, alignment] [; V, alignment] [; criterion] [; CONCODE statement, ..., CONCODE statement])

Note: The optional CONCODE statements will not be discussed in this section. Their format and usage is discussed in Section 1.4.3.

parameters

(i) field name is an unindexed user-defined name.

(ii) character code is either the system name: B, ASCII, EBCDIC, or an unindexed user-defined name.

(iii) length type is either the system name: B or C.

(iv) length is either the string: n, where n is an integer, or the system name NOLIM.

(v) uniformity is either the system name: FIXED (or simply F) or VARIABLE (or simply V).

(vi) data type is either the system name: C, or the string N (base, sign, scale) where:

a) base is the string n, where n is an integer;

b) sign is either the system name: R, RD, NS or the string S (plus, minus) where plus and minus are CONSTANT statements (see Section 1.4.2)

c) scale is either the sytem name FX or the string: FL (E: name, M: name) where name is an unindexed

user-defined name, and either the string E: name or M: name may appear first.

(vii) alignment factor is the string n, where n is an integer.

(viii) and (ix) alignment is the parameter string:

orientation, pad factor

where: orientation is either the system name L or R;

and pad factor is a CONSTANT statement (see Section 1.4.2).

(x) criterion is an unindexed user-defined name.

usage of the statement

The FIELD statement is used to specify a type of field. That is, it specifies the name and implementation for fields in terms of the following parameters:

(i) field name. This parameter gives the name for fields of the type being specified.

(ii) character code. This parameter names the character code to be used in representing fields. The parameter is assigned the system name:

B, when the fields are to be implemented as bit strings,

ASCII, when the ASCII character code is to be used,

EBCDIC, when the EBCDIC code is to be used, and a

user-defined name, when a user-defined character code is to be used. In such a case, the name must appear as the first parameter in a CHAR statement. The use of this option will be discussed in Section 1.4.1.

(iii) length type. The length of a field type is the number of bits or characters that may be used to implement a field of that type. This parameter is used to specify whether length is given in bits or in characters. The parameter is assigned the system name:

B, when length is given in bits, and

C, when length is given in characters of the code specified by parameter (ii).

(iv) length. This parameter is used to specify the number of characters or bits needed to implement a field. The parameter is assigned the string:

n, when not more than n characters or bits are needed, and the system name, and

NOLIM, when there is no limit on the length of the fields.

(v) uniformity. This parameter is used to specify whether all fields of the type being specified are to have the same length. The parameter is assigned the system name:

FIXED (or F), when the length of each field is the same, and

VARIABLE (or V), when the length of each field may be different. In this case, if an integer is specified for length it gives a maximum length for the fields.

(vi) data type. This parameter is used to specify how the fields are to be interpreted. The parameter is assigned the system name:

C, when the fields are to be interpreted as character strings in the character code specified by parameter (i), and the string

N (base, sign, mode) when the fields are to be interpreted as numbers.

a) base. This parameter is assigned an integer which gives the base of the number.

b) sign. This parameter is assigned the system name:

R, when negative numbers are implemented by the radix complement,

RD, when negative numbers are implemented by the diminished radix complement,

S (plus, minus), the CONSTANT statements are used to specify which characters are to be the plus and minus signs,

NS, when no sign is to appear.

c) mode. This parameter is assigned the system name:

FX, when the number is a fixed point number, and

FL (E: name, M: name) when the number is floating point. The exponent and mantissa of the number are described as fields. The names must appear as the first parameters of

FIELD statements. When the string E: name appears first the exponent is to appear before the mantissa, otherwise the mantissa appears first.

(vii) and (viii) F: alignment factor, alignment

alignment factor. This parameter is used in specifying the alignment of fields with respect to arbitrary boundaries (e.g., word, half-word, byte boundaries). The integer specifies the number of bits between alignment boundaries.

alignment. This parameter specifies the actual alignment of the field. The orientation parameter is assigned the system name:

L, when the field is to end on a boundary, and

R, when the field is to begin on a boundary.

The pad factor parameter specifies the characters that are to fill unused storage between the preceding field and the field being aligned.

(ix) V: alignment. This parameter specifies the alignment of the characters or bits of a field when the length is longer or shorter than the number of bits or characters reserved for it. The orientation parameter is assigned the system name:

L, when the field is to be left aligned (truncation or padding occurs at the right), and

R, when the field is to be right aligned (truncation or padding occurs at the left).

The pad factor parameter specifies the characters or bits that are to fill the unused space.

(x) criterion. This optional parameter names a criterion which is to be applied to the fields of the type being specified. If the criterion is not satisfied the field does not occur in the record. The criterion name must appear as the first parameter in a CRITERION statement.

Example 1

Consider a set of fields, each of which is an ASCII character string of variable length. Assume the maximum length of each field is to be 30 characters. The fields are to be referred to by the name 'COLLEGE' and are to be used as names of colleges. The following statement specifies the type for these fields.

```
FIELD ( 'COLLEGE', ASCII, C, 30, V, C )
```

where: 'COLLEGE' is the field name.

ASCII is the character code.

C specifies that length is given in ASCII characters.

30 specifies that length of fields cannot exceed 30 characters.

V specifies that lengths may vary from field to field.

C specifies that the fields are to be interpreted as character strings.

The following character strings are fields of the type speci-

fied in the above statement:

UNIVERSITY OF PENNSYLVANIA

PURDUE

YALE

Example 2

Consider a set of fields, each of which is an ASCII character string of fixed length. Assume the length of each field is 2 characters. The fields are to be referred to by the name 'YEARS' and are to be used as the number of years spent at a college.

The following statement specifies a field for these values:

```
FIELD ( 'YEARS', ASCII, C, 2, F, C )
```

The following character strings are fields o f the type specified in the above statement:

04

02

10

format

GROUP (group name, group order; (list), ... , (list)

[; CONCODE statement, ... , CONCODE statement])

Note: The optional CONCODE statements will not be discussed in this section. Their format and usage are discussed in Section 1.4.3.

parameters

(i) group name is an unindexed user-defined name.

(ii) group order is either the system name: NOORD or SPEC.

(iii) list is a string of parameters with the following format:

name, occurrence, repetition number, repetition uniformity [; O, order] [; V, criterion name]

where:

a) name is an unindexed user-defined name;

b) occurrence is either the system name: MANDATORY (or simply M), or OPTIONAL (or simply O);

c) repetiton number is either the string:
n, where n is an integer, or the system name NOLIM;

d) repetition uniformity is either the system name: FIXED (or simply F), or VARIABLE (or simply V);

e) the optional parameter, order, is either the system name: ASCEND, or DESCND, or an unindexed user-defined name;

f) the optional parameter, criterion name, is an unin-

dexed user-defined name.

usage of the statement

The GROUP statement specifies a type of group. That is, it specifies the structure and implementation of a set of fields and/or other groups*, in terms of the following parameters:

usage of the parameters

(i) group name. This parameter is used to refer to the type of group being specified.

(ii) order. This parameter gives the order in which fields and/or subordinate groups are to occur in the group being specified. The parameter is assigned the system name:

NOORD, when the fields and/or subordinate groups may occur in any order; and

SPEC, when they must occur in the order in which they are named in parameter (iii).

(iii) (list), ..., (list). Each list specifies how a field or subordinate group is to be implemented as part of the group being specified.

a) name. This parameter names the type of a field or subordinate group which is to be included in the group. The name must appear as the first parameter in a FIELD or a GROUP statement.

* Groups included in other groups are called subordinate groups.

b) occurrence. This parameter specifies whether fields or subordinate groups of the type named by parameter (a) are optional or mandatory. The parameter is assigned the system name:

MANDATORY (or M), when the fields or groups of the type named must occur in each group, and

OPTIONAL (or O), when the fields or groups are not mandatory.

c) repetition number. This parameter specifies whether the number of fields or subordinate groups of the type named by parameter (a) are to occur in each group. The parameter is assigned the string:

n, when at most n fields or subordinate groups may occur, and

NOLIM, when any number may occur.

d) repetition uniformity. This parameter specifies whether the parameter, repetition number, gives the exact or the maximum number of fields or subordinate groups. The parameter is assigned the system name:

FIXED (or F), when the repetition number gives the exact number, and

VARIABLE (or V), when it gives the maximum number.

e) order. This parameter specifies the order for fields or subordinate groups of the type named by parameter (a) when more than one of the type is to occur (i.e., the repetition number parameter is NOLIM, or n, greater than 1). The parameter is assigned the system name:

ASCEND, when parameter (a) names a field type whose fields are to be arranged in ascending order;

DESCND, when parameter (a) names a field type whose fields are to be arranged in descending order; and

an unindexed user-defined name, when parameter (a) names a field or group type whose fields or groups are to be arranged sequentially according to a criterion which determines for each pair of fields or groups which is to occur first. The name must appear as the first parameter in a CRITERION statement.

f) criterion name. This parameter names a criterion which determines whether fields or subordinate groups of the type name are to occur. The name must appear as the first parameter in a CRITERION statement.

Example 1

Consider two sets of fields of the following types:

(i) the set of fields of type 'COLLEGE' described in Example 1 of Section 1.1, and

(ii) the set of fields of type 'YEARS' described in Example 2 of Section 1.1.

These fields are to be organized into groups of a type 'COLINF' where one field of type 'COLLEGE' is to be followed by one field of type 'YEARS'. The following statements specify this group type:

```
FIELD ( 'COLLEGE', ASCII, C, 30, V, C )
FIELD ( 'YEARS', ASCII, C, 2, F, C )

GROUP ( 'COLINF', SPEC;
  ( 'COLLEGE', M, 1, F ),
  ( 'YEARS', M, 1, F ) )
```

The following character strings are groups of type 'COLINF':

(i) PURDUE 03

(ii) UNIVERSITY OF PENNSYLVANIA 04

Example 2

Consider three sets of fields of the following types:

(i) a set of fields, each field of which is an ASCII character string of variable length of 5 characters maximum. The fields are to be referred to by the name 'SURNAME' and are to be used as person's name.

(ii) the set of fields of type 'COLLEGE' described in Example 1 of Section 1.1.

(iii) the set of fields of type 'YEARS' described in Example 2 of Section 1.2.

Fields of these types are to be organized into groups of type 'PERSDATA' where one field of type 'SURNAME' is to be followed by zero or more occurrences of fields of types 'COLLEGE' and 'YEARS' organized into groups of type 'COLINF', described in Example 1 above. The following statements specify this group type:

```
FIELD ( 'SURNAME', ASCII, C, 15, V, C )
FIELD ( 'COLLEGE', ASCII, C, 30, V, C )
FIELD ( 'YEARS', ASCII, C, 2, F, C )
GROUP ( 'COLINF', SPEC;
  ( 'COLLEGE', M, 1, F ),
  ( 'YEARS', M, 1, F ) )
```

```
GROUP ( 'PERSDATA', SPEC;
  ( 'SURNAME', M, 1, F ),
  ( 'COLINF', 0, NOLIM, V ) )
```

The following character strings are groups of type 'PERSDATA':

(i) DANIELS

(ii) DANIELS PURDUE 03

(iii) DANIELS PURDUE 03 UNIVERSITY OF PENNSYLVANIA 01

1.3 RECORD Statements

format

RECORD (record name, group name)

parameters

(i) record name is an unindexed user-defined name.

(ii) group name is an unindexed user-defined name.

usage of the statement

The RECORD statement is used to specify that a type of group is to be treated as a record, i.e., it is to be used as a basic unit of storage and retrieval.

(i) record name. This parameter gives the name for records of the type being specified.

(ii) group name. This parameter is the name of the type of group which is to be treated as a record.

Example

Consider groups of type 'PERSDATA' described in Example 2 of Section 1.2.

The following statement indicates that these groups are to be treated as records (called 'PERSRCD'):

```
RECORD ( 'PERSRCD', 'PERSDATA' )
```

The complete description for this type of record, then, is given by the following statements:

```
FIELD ( 'SURNAME', ASCII, C, 15, V, C )
FIELD ( 'COLLEGE', ASCII, C, 30, V, C )
FIELD ( 'YEARS', ASCII, C, 2, V, C )
GROUP ( 'COLINF', SPEC;
   ( 'COLLEGE', M, 1, F ),
```

```
      ( 'YEARS', M, 1, F ) )
   GROUP ( 'PERSDATA', SPEC;
      ( 'SURNAME', M, 1, F ),
      ( 'COLINF', 0, NOLIM, V ) )
   RECORD ( 'PERSRCD', 'PERSDATA' )
```

The following groups are records of the type specified by the above GDDL statements:

i) DANIELS

ii) DANIELS PURDUE 03

iii) DANIELS PURDUE 03 YALE 01

1.4 Record Specification Substatements

The Record Specification substatements are statements which are primarily used as parameters in Record Specification statements or are referred to by Record Specification statements.

1.4.1 CHAR Statements

format	CHAR (character code name; set name, set name)
parameters	(i) character code name is an unindexed user-defined name. (ii) and (iii) set name is an unindexed user-defined name.
usage of the statement	The CHAR statement is used to specify a character code for representing values. The new character code is specified in terms of an existing one.
usage of the parameters	(i) character code name. This parameter gives the name used in referring to the character code being specified. (ii) and (iii) set name, set name. These parameters specify the character code. They refer to SET statements which list the binary code for each character, give the sort order for the characters and relate the characters to their corresponding codes in an existing character code. This latter specification is necessary during conversions when fields are converted from one character code to another. Each name must appear as the first parameter in a SET statement. The use of the SET statement in specifying a character code is described in Section 2.1.2.1.

1.4.2 CONSTANT Statements

format

CONSTANT (character string, character code)

parameters

(i) character string is any string of characters or bits. If the string contains the characters (or), then they must be surrounded by apostrophes.

(ii) character code is either the system name B, ASCII, EBCDIC, or a user-defined name.

usage of the statement and parameters

The CONSTANT statement is used as a parameter when arbitrary character strings are required. The system name CONSTANT and the parentheses serve to delimit the character string for the DDL processor. For this reason, if parentheses must occur in the character string, then they must be surrounded by apostrophes.

The character code must be assigned the system name:

B, when the string contains only the characters 0 and 1 and these are to be interpeted as bits;

ASCII, when the string is to be interpreted as a string of ASCII characters;

EBCDIC, when the string is to be interpreted as a string of EBCDIC characters; and

a user-defined name, when the string is to be interpreted as a string of characters over a user-defined character code. This name must appear as the first parameter in a CHAR statement.

Example

Consider the ASCII string: A(1). If this string is to be used for a parameter, it must be entered as:

```
CONSTANT ( A'('1')', ASCII )
```

Note: Any blanks other than the first blank following the open parenthesis will be considered part of the character string.

1.4.3 CONCODE Statements

format

CONCODE (delimiter, position)

parameters

(i) delimiter is a CONSTANT statement with the format described in Section 1.4.2.

(ii) position is either the system name PRX, PTX, or INX.

usage of the statement

The CONCODE statement is used as a parameter in other GDDL statements to specify delimiters in the form of character strings. These delimiters can be used in records to determine the beginnings or ends of variable length fields, to separate repeating groups and fields, or to identify optional groups or fields.

usage of the parameters

(i) delimiter. The CONSTANT statement gives the character string which is to be used as the delimiter.

(ii) position. This parameter is used to specify the position of the delimiter relative to the group or field being delimited. It is assigned the system name:

PRX, when the delimiter is to precede (prefix) the field or group,

PTX, when the delimiter is to follow (postfix) the field or group, and

INX, when the delimiter is to be inserted between (infix) repeating fields or groups.

Example 1

Consider the field type 'COLLEGE' specified in Example 1 of Section 1.1:

Each field of that type is an ASCII character string of variable length. The maximum length is 30 characters. A delimiter such as a comma followed by a blank can be used to indicate the end of the field. The following statement specifies a field type with such a delimiter:

```
FIELD ( 'COLLEGE-NAME', ASCII, C, 30, V, C;
   CONCODE ( CONSTANT ( , , ASCII ), PTX ) )
```

where PTX indicated that the characters, and a blank will follow each field of type 'COLLEGE-NAME'.

The following character strings are fields of the type specified in the above GDDL statement:

i) UNIVERSITY OF PENNSYLVANIA,

ii) PURDUE,

iii) YALE,

Example 2

Consider two field types:

i) the field type 'COLLEGE-NAME' specified in Example 1.

ii) the field type 'YEARS' specified in Example 2 of Section 1.1:

Each field of that type is an ASCII character string of a fixed length of 2 characters.

Fields of these two types are to be organized into groups of a type 'COLLINF' such that, one field of type 'COLLEGE-NAME' is to be followed by one field of type 'YEARS'. Groups of type 'COLLINF' are to be organized as repeating groups into a group of type 'COLLDATA' such that the 'COLLINF' groups are separated from each other by a semicolon and a blank. The following statements specify the group type 'COLLDATA':

```
FIELD ( 'COLLEGE-NAME', ASCII, C, 30, V, C;
  CONCODE ( CONSTANT ( , , ASCII ), PTX ) )
FIELD ( 'YEARS', ASCII, C, 2, F, C )
GROUP ( 'COLLINF', SPEC;
   ( 'COLLEGE-NAME', M, 1, F ),
   ( 'YEARS', M, 1, F );
  CONCODE ( CONSTANT ( ; , ASCII ), INX ) )
GROUP ( 'COLLDATA', SPEC;
   ( 'COLLDATA', M, NOLIM, V ) )
```

The following character strings are groups of type 'COLLDATA':

i) PURDUE, 03

ii) PURDUE, 03; YALE, 01

iii) UNIVERSITY OF PENNSYLVANIA, 04; PURDUE, 03; YALE, 01

1.4.4 Parameter Statements

Parameter statements are statements that are used as parameters in other GDDL statements. They may be used to specify any parameter. These statements are used when the user wants a parameter to depend on fields and/or other characteristics of his data structure. For example, a user may want to have the length of a field equal to the number of times another field repeats in a record. Parameter statements allow such a relationship to be described.

When a parameter is to depend on a single field, this is specified by replacing the parameter with the name of the field. Similarly, when a parameter is to depend on a single other characteristic, this is specified by replacing the parameter with either the LENGTH, COUNT, or PARAMVAL statements described in the following sections. When a parameter is to depend on a function of fields and characteristics, the PARAMPROG statement of Section 1.4.4.4 is used.

Since more than one field or group of a particular type may appear in a record (by repetition or by inclusion in different groups), specific occurrences of the field or group type must be referred to in a parameter statement. Such referencing is done by modifying the field or group name in the following ways:

(i) if the field or group of type 'X' repeats, the n^{th} value of the field, or n^{th} values of the group, is referred to by the indexed name: 'X(n)' where n is an integer.

(ii) if the field or group of type 'X' occurs in more than one group, say in groups 'T', 'U', and 'V', then the values of the field

or group of type 'X' are referred to by the name: 'X' OF 'U'. As many of the phrases "OF group name" may be specified as are necessary to distinguish between different values of the field or group type. For example, in the above case, if the group of type 'U' occurs in groups of types 'S' and 'R', then the name:

'X' OF 'U' OF 'S'

refers to the values of 'X' which occur in 'U' which occur in 'S'.

In defining different organizations of fields, it will occasionally be necessary to indicate the record and the organization of records (files) in which values of a field or group occur. Such referencing is done by modifying the field or group name in the following ways:

(iii) if the field or group of type 'X' occurs in a record 'RCD1', then the field or group is referred to by the name:

'X' OF 'RCD1'

If the field or group occurs in one or more groups, the group in question, say of type 'U', is specified before the record name:

'X' OF 'U' OF 'RCD1'

(iv) if the record, 'RCD1', containing a field or group, 'X', is organized in a file named 'Z', then the field or group is referred to by the name:

'X' OF 'RCD1' OF 'Z'

(v) in general, whenever a structure, 'S1', occurs as part of another structure, 'S2', then the structure 'S1' can always be referred to by the name:

'S1' OF 'S2'

Names, as described in (i), (ii), (iii), (iv) and (v) above, are called reference names.

1.4.4.1 LENGTH Statements

format	LENGTH (data name, length type)
parameters	(i) data name is a reference name. (ii) length type is either the system name: B, ASCII, EBCDIC, or an unindexed user-defined name.
usage of the statement	The LENGTH statement may be used as a parameter when the parameter can be an integer (e.g., the length parameter in the FIELD statement, or the repetition number parameter in the GROUP statement). The LENGTH statement is used to assign the length of the occurrence of a structure such as a field or group to a parameter.
usage of the parameters	(i) data name. This parameter refers to a structure whose length is to be used as a parameter. (ii) length type. This parameter specifies whether the length is to be given in terms of bits or characters. The parameter is assigned the system name: B, when length is to be given in bits, ASCII, or EBCDIC when the length is to be given in ASCII or EBCDIC characters, and a user-defined name, when the length is to be given in characters of a user-defined character code. The name must appear as the first parameter in a CHAR statement.

Example

Consider a field type 'COLLEGE' whose fields have variable lengths of up to 30 ASCII characters maximum. Assuming that fields of this type occur only once in a record, another field type (say 'X') may be specified for the record, whose fields are to have the same number of bits in length as 'COLLEGE' fields have in characters. That is, if a field of type 'COLLEGE' has a length of n characters in a record, a field of type 'X' in that record has a length of n bits.

The following statement specifies such a field type:

```
FIELD ( 'X', B, B, LENGTH ( 'COLLEGE', ASCII ), V, C )
```

1.4.4.2 COUNT Statements

format

COUNT (data name)

parameter

data name is a reference name

usage of the statement and parameter

The COUNT statement may be used as a parameter when the parameter can be an integer. The COUNT statement is used when the parameter is to be assigned the number of times a structure such as that of a field type, referred to by the data name, occurs. If the structure does not occur, the parameter is assigned the number 0.

Example

Consider a group type 'COLINF' which may occur zero or more times in a record. The user may define a field type (say 'Y') whose fields are to be interpreted as ASCII character strings with their lengths in characters equal to the number of

occurrences of 'COLINF'.

The following statement specifies this field type:

```
FIELD ( 'Y', ASCII, C, COUNT ( 'COLINF' ), V, C )
```

Thus, in a record if 'COLINF' repeats twice, then a field of type 'Y' will have a length of 2 ASCII characters.

1.4.4.3 PARAMVAL

format	PARAMVAL (name, parameter number)
parameters	(i) name is a reference name. (ii) parameter number is the string n, where n is an integer.
usage of the statement	The PARAMVAL statement may be used as a parameter when the parameter is to depend on a parameter in another GDDL statement.
usage of the parameters	(i) name. This parameter identifies the statement whose parameter is to be used. The name is the one that appears first in the statement. (ii) parameter number. This parameter identifies which parameter in the statement identified by parameter (i) is to be used.
Example	Consider a field type 'X' described as follows: FIELD ('X', ASCII, C, NOLIM, V, C)

If the lengths of fields of a type 'Y' are to equal the lengths of fields of type 'X', this is specified as follows:

```
FIELD ( 'Y', ASCII, C, PARAMVAL ( 'X', 4), F, C )
```

The LENGTH statement can also be used to specify this relationship. In general, this statement is meant to describe relationships that are not covered by the previous parameter statements.

1.4.4.4 PARAMPROG Statements

format

PARAMPROG (program name; parameter, ..., parameter)

parameters

(i) program name is an unindexed user-defined name.

(ii) parameter is either:

a reference name,

a parameter statement, or

a CONSTANT statement.

usage of the statement

The PARAMPROG statement may be used as a parameter when that parameter is to be a function of the other parameters, values and/or constants.

usage of the parameters

(i) program name. This parameter gives the name of a program supplied by the user to compute the function required.

(ii) parameter, ..., parameter. These parameters identify the fields, parameters and constants for which the function is computed.

Example

Consider a program, ACCUM, which inputs two values, A and B and computes the value of A + B.

The following statement specifies that fields of type 'Z' have lengths equal to the sum of a field of type 'A' and a field of type 'B':

```
FIELD ( 'Z', ASCII, C, PARAMPROG ( 'ACCUM', 'A', 'B' ),
     V, C )
```

2. File Specification Statements

The following terminology will be used in presenting the File Specification statements:

(i) A record is a group which is to be used as a basic unit for storage and retrieval.

(ii) Two records are of the same type if and only if their organization and implementation are specified by the same RECORD statement.

(iii) There is a direct access path from a record (say, A) to another record (say, B) if and only if

a) record B is positioned immediately after record A;

b) record A contains a pointer to record B; or

c) in a table of pointers, a pointer to record B is positioned immediately after a pointer to record A.

Relative to this access path, record A is called the head record, and record B is called the tail record.

(iv) Consider two records (say, C and D). There is an access path of length n from C to D if and only if there are n-1 records $R_1,\ldots,R_{n-1}$ such that there are direct access paths from C to R_1, from R_{n-1} to B, and from R_i to R_{i+1} for $1 \leq i \leq n-1$.

(v) Criteria over fields, access paths and characteristics of data are used to determine when a direct access path is to exist from one record to another.

(vi) Two direct access paths are of the same type if and only if they satisfy the same criteria and are implemented in the same way.

(vii) A file is a set of records with the direct access paths between them.

The File Specification statements are used to specify a file in terms of criteria for determining access paths and the implementation of the access paths.

The statements for specifying criteria are presented in Section 2.1. The LINK Section 2.2 is used to specify the implementation of direct access paths. The FILE statement of Section 2.3 specifies what different types of access paths are to exist for a particular set of records.

2.1 Criteria Statements

Criteria are used to determine when a direct access path is to exist. Criteria can be defined over fields, characteristics of data (as specified by the parameters of GDDL statements), and existing access paths.

In describing such criteria, it may be necessary to indicate that fields, groups or characteristics of fields or groups from different records of the same type are being compared. To refer to such fields, groups or characteristics unambiguously, their reference names are modified in the following way:

The record name following the system name "OF" in the reference name is replaced by the OCC statement of Section 2.1.2.2.

Reference names modified in this way are also called reference names.

When a criterion is defined in terms of records other than the head or tail record, it is necessary to specify whether:

a) the criterion is satisfied, if it is satisfied for all records other than the head and tail records, or

b) the criterion is satisfied, if it is satisfied for at least one record other than the head and tail records.

To specify these cases, the ALLOC and SOMEOCC statements of Sections 2.1.2.3 and 2.1.2.4 are used.

2.1.1 CRITERION Statements

format

CRITERION (criterion name, criterion expression)

parameters

(i) criterion name is an unindexed user-defined name.

(ii) criterion expression is a string of the form:

a) (arithmetic expression) relation symbol (arithmetic expression) where:

1) relation symbol is either the system name:

EQ, or = ,

NQ,

LT, or < ,

LE, or ≤ ,

GT, or > ,

GE

2) arithmetic expression is a string of the form:

i) reference name;

ii) Parameter statement (see Section 1.4.4);

iii) CONSTANT statement (see Section 1.4.2); and

iv) PATH (record reference; record reference; link name) where:

1) record reference is a string of the form:

record name [, criterion name]

where: record name is a reference name, and criterion name is an unindexed user-defined name.

2) link name is an unindexed user-defined name.

v) (arithmetic expression) arithmetic operator (arithmetic expression) where arithmetic operator is either the string: +, -, x, or /.

b) (data name) MEM (set name)

where: 1) data name is a reference name, and

2) set name is an unindexed, user-defined name.

c) NOT (criterion expression form)

where criterion expression form is either an unindexed user-defined name, or a criterion expression.

d) (criterion expression form) AND (criterion expression form)

where criterion expression form is defined as in c) above.

e) (criterion expression form) OR (criterion expression form)

where criterion expression form is defined as in c) above.

f) ALLOCC statement. This form will be discussed in Section 2.1.2.3.

g) SOMEOCC statement. This form will be discussed in Section 2.1.2.4.

usage of the statement

Criteria on fields, characteristics and access paths are used for three purposes:

(1) to determine when records are to be linked by access paths;

(2) to determine when particular fields and groups which are optional in a record are to occur (see the FIELD statement, parameter (x) and the GROUP statement, parameter (iii) f); and

(3) to identify records for data conversion (see Section 3).

A criterion is specified in terms of the following parameters:

usage of the parameters

(i) criterion name. This parameter gives the name to be used in referring to the criterion being specified.

(ii) criterion expression. This parameter gives the criterion. Each form this parameter may take will be discussed separately.

a) (arithmetic expression) relation symbol (arithmetic expression)

This form is used to specify that if the evaluation of an arithmetic expression is:

EQ, or =, equal to,

NQ, not equal to,

LT, or <, less than,

LE, less than or equal to,

GT, or >, greater than, or

GE, greater than or equal to

the evaluation of a second arithmetic expression, then the criterion will be satisfied.

An arithmetic expression is given by

i) a reference name when the evaluation of the expression is to be the value of a field.

ii) a parameter statement when the evaluation of the expression is to be the parameter for a particular characteristic which had been specified as depending on other values, etc. or as varying.

iii) a CONSTANT statement when the evaluation is to be a particular constant.

iv) an expression of the form

PATH (record reference; record reference;
link name)

when the evaluation of the expression is to be the length of a path. If the path specified by the expression does not exist, the evaluation of the expression is, by convention, zero. The parameters in this form are used as follows:

1) record reference. This parameter specifies the record at which the path is to begin. Record name gives the record type (it may also name a specific record of the type, see Section 2.1.2.2). Criterion name gives the name of a criterion used to select the particular record desired.

2) record reference. This second parameter specifies the record at which the path is

to end. This is done in a similar way to the specification in parameter 1) above.

3) link name. This parameter specifies the type of link which determines the path to be tested. Link name must appear as the first parameter in a LINK statement (see Section 2.2).

v) an expression of the form

(arithmetic expression) arithmetic operator (arithmetic expression)

when the evaluation of the expression is to be the evaluation of the first arithmetic expression plus (+), minus (-), times (x), or divided by (/) the evaluation of the second arithmetic expression.

b) (data name) MEM (set name)

This form is used to specify that the value of a field referred to by the data name is a member of the set referred to by set name. Set name must appear as the first parameter in a SET statement (see Section 2.1.2.1).

c) NOT (criterion expression form)

This form is used to specify that the negation of a criterion expression is to be tested. The criterion expression may be given directly or named.

If the criterion expression is named, the name must appear as the first parameter in another CRITERION statement.

d) (criterion expression form) AND (criterion expression form)

This form is used to specify that the logical conjunction of two expressions is to be tested. The criterion expressions may be given directly or named. If the criterion expressions are named, the names must appear as the first parameters in other CRITERION statements.

e) (criterion expression form) OR (criterion expression form)

This form is used to specify that the logical disjunction of two expressions is to be tested. The criterion expressions may be given directly or named. If the criterion expressions are named, the names must appear as the first parameters in other CRITERION statements.

Example 1

Consider an optional field 'DRAFT STATUS' in a record of type 'PERSNL'. A user may test to see if this field occurs by specifying the following criterion (called 'TEST1'):

```
CRITERION ( 'TEST1', ( EXIST ( 'DRAFT STATUS' OF
        'PERSNL' )) EQ ( CONSTANT ( 1, ASCII )))
```

Example 2

Consider a record of type 'PERSNL' containing three fields: 'YEARS', 'STARTDATA', and 'ENDDATE', each of which is a numeric field. A user may test to see if the value of 'YEARS' equals the difference between the values of 'ENDDATE' and 'STARTDATE' by specifying the following criterion (called 'TEST2'):

```
CRITERION ( 'TEST2', ( 'YEARS' OF 'PERSNL' )  EQ
      ( ( 'ENDDATE' OF 'PERSNL' ) - ( 'STARTDATE' OF
          'PERSNL' )))
```

Example 3

Consider the criterion described in Example 2, above. The criterion which is the negation of this can be specified in two ways (called 'TEST3.1' and 'TEST3.2').

```
CRITERION ( 'TEST3.1', NOT ( 'TEST2' ))
```

```
CRITERION ( 'TEST3.2', NOT ( ( 'YEARS' OF 'PERSNL' )
      EQ ( ( 'ENDDATE' OF 'PERSNL' ) -
       ( 'STARTDATE' OF 'PERSNL' ))))
```

2.1.2 Criterion Substatements

The Criterion substatements are statements which either are used as parameters in Criterion statements or are referred to primarily by Criterion statements.

2.1.2.1 SET Statements

format	SET (set name; member, ..., member)
parameters	(i) set name is a user-defined name. (ii) member is a CONSTANT statement.
usage of the statement	The SET statement is used to specify a set of strings over some character code. This set is specified by listing each member and assigning a name to the collection. SET statements are used together with CRITERION statements (see Section 2.1.1) to specify set-theoretic criterion expressions. SET statements may also be used to define character codes (see Section 1.4.1). In this case two SET statements are required. The first SET statement is used to specify the individual character codes as bit strings. These are listed in their sort order. The second SET statement is used to relate these bit strings to their equivalents in an existing code. This is done by listing the bit strings of the existing code in the same order as their equivalents in the new code were listed. If there are fewer bit strings in the existing code, then the string *** is used in place of the lacking bit string.

usage of the parameters

(i) set name. This parameter is the name of the set being specified.

(ii) member. A CONSTANT statement is used to specify each member of the set being defined.

Example

Consider a set of character strings: BURNS, JACOBS, MILLER, and SANDERSON. These are to be interpreted as authors' names. The following statement assigns the name 'AUTHORS' to this set.

```
SET ( 'AUTHOR'; CONSTANT ( BURNS, ASCII ),
   CONSTANT ( JACOBS, ASCII ), CONSTANT
   ( MILLER, ASCII ), CONSTANT ( SANDERSON, ASCII ) )
```

The following criterion (called 'TESTSET') tests whether the value of a field of type 'AUTH' in a record of type 'BOOK' is a member of this set:

```
CRITERION ( 'TESTSET', ( 'AUTH' OF 'BOOK' )
  MEM ( 'AUTHOR' ) )
```

2.1.2.2 OCC Statements

format	OCC (record name, occurrence name)
parameters	(i) record name is an unindexed user-defined name. (ii) occurrence name is either the system name: T, or H, or record variable name, where this name has the form Xn, where n is an integer.
usage of the statement	The OCC statement is used in a reference name to identify a particular record of a given type in terms of the following parameters:
usage of the parameters	(i) record name. This parameter names the type of record being referred to. (ii) occurrence name. This parameter specifies the particular record, that is being referred to. The parameter is assigned the system name: T, if the record is the tail record of the direct access path. H, if the record is the head record of the direct access path, and Xn, if the record is some record other than the head or tail records. For each such distinct record the integer, n, must be different.

Example 1

Consider the case where a type of direct access path is being defined between records of a particular type, say 'PERSRCD', which is described in the Example of Section 1.1.3. If the criterion (say 'CRITEX1') determining the access paths is defined in terms of the field 'SURNAME' in 'PERSRCD', such that the value of 'SURNAME' in the head record is less than or equal to the value of 'SURNAME' in the tail record, then the following statements are used to specify each of these particular records:

```
OCC ( 'PERSRCD', T )
OCC ( 'PERSRCD', H )
```

These statements are used in referenc names to refer to the value of the field 'SURNAME' in each 'PERSRCD' record as follows:

```
'SURNAME' OF OCC ( 'PERSRCD', T )
'SURNAME' OF OCC ( 'PERSRCD', H )
```

These reference names would be used in a criterion expression, to specify the criterion 'CRITEX1', as follows:

```
CRITERION ( 'CRITEX1',
    ( 'SURNAME' OF OCC ( 'PERSRCD' H ) )
    LE ( 'SURNAME' OF OCC ( 'PERSRCD', T ) )
```

Example 2

Consider the case where a type of direct access path is being defined between records of the type 'PERSRCD' as in Example 1, above. In this case the criterion determining the access paths is defined in terms of 'SURNAME' fields such that two records are linked if and only if the values of 'SURNAME' in the head and tail records are related as in Example 1, above, and in addition there is no other 'PERSRCD' record in which the value of 'SURNAME' is less than the value of 'SURNAME' in the tail record and greater than the value of 'SURNAME' in the head record. To describe this by a criterion expression, reference must be made to 'PERSRCD' records other than the head and tail records. The following statement specifies this:

```
OCC ( 'PERSRCD', X1 )
```

This statement would be used in a reference name to refer to the value of the field 'SURNAME' in the other 'PERSRCD' records as follows:

```
'SURNAME' OF OCC ( 'PERSRCD', X1 )
```

This reference name would be used in a criterion expression to specify the part of the criterion relating to the other 'PERSRCD' records as follows:

```
NOT ( ( ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) )
    LT ( 'SURNAME' OF OCC ( 'PERSRCD', T ) ) )
    AND ( ( 'SURNAME' OF OCC ( 'PERSRCD', H ) )
    LT ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) ) ) )
```

Note: To completely specify this part of the criterion the criterion expression above would appear as a parameter in an ALLOCC statement. This is demonstrated in the example given in Section 2.1.2.3 below.

2.1.2.3 ALLOCC Statements

format	ALLOCC (record variable name, ..., record variable name; criterion expression)
parameters	(i) record variable name is a string of the form Xn, where n is an integer. (ii) criterion expression in this statement is the same as that defined for parameter (ii) of the CRITERION statement.
usage of the statement	The ALLOCC statement is used as a criterion expression in a CRITERION statement. An ALLOCC statement indicates that the criterion expression in the statement is satisfied when it is true for all records identified in terms of the record variable names given by the first parameters of the statement.
usage of the parameters	(i) record variable name, ... record variable name. These parameters specify that all records (other than the head and tail records) of the type, identified via the record variable name in the corresponding OCC statement, must be tested to determine if the criterion expression in which they appear is satisfied.

(ii) criterion expression. This parameter specifies the criterion expression. It must contain at least one OCC statement having a second parameter of the form Xn for each record variable name of the form Xn in parameter (i).

Example

To completely specify the criterion described in Example 2 of Section 2.1.2.2, it is first necessary to state that all records of type 'PERSRCD' must be tested to determine if the criterion expression is satisfied. This is specified by the following statement:

```
CRITERION ( 'CRITEX2', ALLOCC ( X1;
    NOT (  ( ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) )
    LT ( 'SURNAME' OF OCC ( 'PERSRCD', T ) )
    AND  ( ( 'SURNAME' OF OCC ( 'PERSRCD', H ) )
    LT ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) ) ) ) ) )
```

Then, this criterion must be combined with the criterion 'CRITEX1' to form the conjunction:

```
CRITERION ( 'CRITEX3', ( 'CRITEX1' ) AND ( 'CRITEX2' ) )
```

2.1.2.4 SOMEOCC Statements

format

SOMEOCC (record variable name, ..., record variable name;
criterion expression)

parameters

(i) record variable name is a string of the form Xn, where n is an integer.

(ii) criterion expression in this statement is the same as that defined for parameter (ii) of the CRITERION statement.

usage of the statement

The SOMEOCC statement is used as a criterion expression in a CRITERION statement. A SOMEOCC statement indicates that the criterion expression is satisfied when it is true for at least one record identified in terms of the record variable names given by the first parameters of the statement.

(i) record variable name, ..., record variable name. These parameters specify that at least one record of the type (other than the head and tail records) identified, via the record variable name in the corresponding OCC statement, must satisfy the criterion expression in which it is named, if the criterion expression is to be satisfied.

(ii) criterion expression. This parameter specifies the criterion expression. It must contain at least one OCC statement having a second parameter of the form Xn for each record variable name of the form Xn in

parameter (i).

Example

The criterion 'CRITEX2', described in the Example of Section 2.2.2.3, can also be described by stating that: it must not be the case that there is a single record of the type 'PERSRCD' which contains a value of 'SURNAME' that is greater than the value of 'SURNAME' in the head record and less than the value of 'SURNAME' in the tail record:

```
CRITERION ( 'CRITEX2-ALT',
 NOT ( SOMEOCC ( X1;
   ( ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) )
   LT ( 'SURNAME' OF OCC ( 'PERSRCD', T ) )
   AND  ( ( 'SURNAME' OF OCC ( 'PERSRCD', H ) )
   LT ( 'SURNAME' OF OCC ( 'PERSRCD', X1 ) ) ) ) ) ) )
```

2.2 LINK Statement

format

```
LINK ( link name; head record name, tail record name;
   criterion name, implementation; link number, link
   uniformity )
```

parameters

(i) link name is an unindexed user-defined name.

(ii) head record name is an unindexed user-defined name.

(iii) tail record name is an unindexed user-defined name.

(iv) criterion name is an unindexed user-defined name.

(v) implementation is either the system name:

SEQUEN; or the string

EMBED, pointer name [; P, length] [; R, count]

where pointer name is an unindexed user-defined name, and length and count are strings of the form n, where n is an integer.

DIREC, pointer name, file name

where pointer name and file name are unindexed user-defined names.

(vi) link number is either the string n, where n is an integer, or the system name NOLIM.

(vii) link uniformity is either the system name:

FIXED (or simply F), or

VARIABLE (or simply V).

usage of the statement

The LINK statement is used to specify a type of direct access path. That is, it specifies the criterion, which determines when direct access paths are to exist, and the implementation of these paths in terms of the following parameters:

(i) link name. This parameter gives the name for direct access paths of the type being specified.

(ii) head record name. This parameter is used to name the record type for the head record of the direct access path. The name must appear as the first parameter in a RECORD statement.

(iii) tail record name. This parameter is used to name the record type for the tail record of the direct access path. The name must appear as the first parameter in a RECORD statement.

(iv) criterion name. This parameter is used to name the criterion which determines when direct access paths are to exist between two records. The name must appear as the first parameter in a CRITERION statement.

(v) implementation. This parameter is used to specify the implementation of the direct access paths. Implementation is given by the system name:

SEQUEN - when the tail record is to be stored sequentially after the head record of the path.

EMBED, pointer name [; P: length] [; R: count]
when there is to be a pointer to the tail record stored in the head record.

The parameter, pointer name, gives the name of the pointer. This name must appear as the first parameter in a POINTER statement (see Section 3.3).

The optional parameter, length, is used when a maximum is to be placed on the length of any path implemented by the pointers.

The optional parameter, count, is used when a maximum is to be placed on the number of records linked together by the pointers.

DIREC, pointer name, file name - when a pointer to the tail record is to be stored in a table after a pointer to the head record.

The parameter, pointer name, gives the name of the pointer. This name must appear as the first parameter in a POINTER statement (see Section 3.3).

The table must be described as a set of records structured in some way by access paths. The parameter, file name, names the specification of the table. This name must appear as the first parameter in a FILE statement (see Section 2.3).

(vi) link number. This parameter specifies the number of tail records that may be linked by direct access paths of the type currently specified to each head record.

The parameter must be assigned the string n, when either exactly n tail records, or a maximum of n tail records, are to be linked to each head record.

The parameter is assigned the system name NOLIM, when there is no limit on the number of tail records linked to each head record.

(vii) link uniformity. This parameter specifies whether the link number of parameter (vi) gives the exact number of tail records linked to each head record. The parameter is assigned the system name:

FIXED (or F) when there may be a fixed number of tail records linked to each head record. In this case link number gives the exact number.

VARIABLE (or V) when there may be a variable number of tail records linked to each head record. In this case the link number gives the maximum number of the tail records.

Example

The following statement specifies a type of direct access path (called 'LINK1'). These paths link records of type 'PERSRCD' (as specified in Example 1 of Section 1.3) and are determined by the criterion 'CRITEX3' specified in the example of Section 2.1.3.3. The statement determines a list structure for the records which is implemented by sequential stroage.

```
LINK ( 'LINK1'; 'PERSRCD', 'PERSRCD';
    'CRITEX3', SEQUEN; 1, FIXED )
```

2.3 FILE Statement

format

FILE (file name; link name, ..., link name;
storage specification name; device specification name, ...,
device specification name.

parameters

(i) file name is an unindexed user-defined name.

(ii) link name is an unindexed user-defined name.

(iii) storage specification name is an unindexed user-defined name.

(iv) device specification name is an unindexed user-defined name.

usage of the statement

The FILE statement is used to specify a file by giving the different types of access paths that exist among the records and by relating this specification with a particular storage structure. A separate FILE statement must be provided for each set of records that is to be treated as a table for pointers (see Section 2.2, parameter (iv) - the DIREC option).

usage of the parameters

(i) file name. This parameter gives the name used in referring to the file.

(ii) link name, ..., link name. These parameters give the names of the access path which can exist in the file. Each name must appear as the first parameter in a LINK statement. There must be at least one LINK

statement for each record type which specifies a criterion for sequencing that type of record relative to one of the other types in the file.

(iii) storage specification name. This parameter specifies which storage structure is associated with the file.

(iv) device specification name, ..., device specification name. These parameters specify the devices on which the file is to be stored.

3. Storage Specification Statements

Storage devices are organized by arranging certain storage units into a hierarchy, so that the storage unit at each level of the hierarchy (with the exception of the lowest level) can be decomposed into various storage units at the next lower levels. For example, the magnetic disk consists of the following storage units: physical blocks, tracks, cylinders, and disk packs. These storage units are arranged in a hierarchy in that physical blocks are organized into tracks, tracks are organized into cylinders, and cylinders are organized into a particular disk pack.

The following terminology will be used in presenting the Storage Specification statements:

(i) A basic block is the storage unit at the lowest level of a storage device (e.g., the physical blocks of a disk). They consist of storage positions such as card columns on a punched card, bits on a disk, or 7 or 9-bit characters on a tape.

(ii) A block is the storage unit at any of the higher levels of a storage device (e.g., the tracks or cylinders of a disk). Thus, a block is an organization of basic blocks and/or other blocks. We call this organization of blocks the storage structure.

(iii) Two blocks or basic blocks are of the same type if and only if they have the same structure and implementation.

The Storage Specification statements are used to specify:

(1) the storage structure for a file,

(2) the implementation of this storage structure,

(3) the positioning of the records of the file within the storage

structure, and

(4) the form of the record addresses which are to be used as pointers in the file.

3.1 BBLOCK Statements

format

BBLOCK (basic block name; length, uniformity;
record count, basic block count, count uniformity
[; SPLIT: record name, ..., record name]
[; START: record name, ..., record name]
[; HDR: header] ... [; HDR: header]
[; TLR: trailer] ... [; TLR: trailer])

parameters

(i) basic block name is an unindexed user-defined name.

(ii) length is either the string n, where n is an integer, or the system name NOLIM.

(iii) length uniformity is either the system name:
FIXED (or simply F), or
VARIABLE (or simply V).

(iv) record count is either the string n, where n is an integer, or the system name NOLIM.

(v) basic block count is either the string n, where n is an integer, or the system name NOLIM.

(vi) count uniformity is either the system name:
FIXED (or simply F), or
VARIABLE (or simply V).

(vii) and (viii) record name is an unindexed user-defined name.

(ix) header is either a CONSTANT statement with the format described in Section 1.4.2, or an unindexed user-defined name.

(x) trailer is either a CONSTANT statement with the format described in Section 1.4.2, or an unindexed user-defined name.

usage of the statement

The BBLOCK statement is used to specify a type of basic block in the storage structure and the positioning of records within that type of basic block. The statements of Section 3.4 determine whether such basic blocks are to be interpreted as a punched card, physical tape block, etc.

usage of the parameters

(i) basic block name. This parameter gives the name for basic blocks of the type being specified.

(ii) length. This parameter gives the number of storage positions in a basic block. The parameter is assigned the string:

n, when the length of the basic block is to be at most n storage positions; and

NOLIM, when there is to be no limit on the length of the basic block.

(iii) length uniformity. This parameter specifies whether every basic block of the type being specified is to have the same length. The parameter is assigned the system name:

FIXED (or F), when the length of all basic blocks are the same; and

VARIABLE (or V), when the length of each basic block is to differ. In this case, the length gives the maximum number of storage positions in each basic block.

(iv) record count, and

(v) basic block count. These parameters specify the number of records stored in a basic block as a ratio of records per basic block. The two parameters are assigned the strings:

n, m - when n records (or at most n records) are to be placed in m basic blocks.

1, NOLIM - when single records of varying length may exceed the length of a basic block, and each record is to begin in a new basic block.

NOLIM, 1 - when there is no limit on the number of records to be placed in a single basic block.

(vi) count uniformity. This parameter specifies whether the number of records per basic block is fixed. The parameter must be assigned the string:

FIXED (or F), when the number of records per block is fixed, and

VARIABLE (or V), when the number of records per block is to vary. In this case, record count gives the maximum number of records.

(vii) SPLIT: record name, ..., record name. These parameters specify those record types which may be split between basic blocks when there are not enough storage positions for an entire record. Each record name must appear as the first parameter in a RECORD statement.

(viii) START: record name, ..., record name. These parameters are used to specify those record types which may occur first in a basic block. Each record name must appear as the first parameter in a RECORD statement. If no record names are specified, the basic block is to contain no records of the file.

(ix) HDR: header. This optional parameter is used to specify control information which occurs at the head of a basic block. The statements describing the individual devices (see Section 3.4) specify whether the header is to be interpreted as a start card for a card deck, or a label for a tape, etc. The parameter is assigned:

a CONSTANT statement, when the character string contained in the header is to be given directly, and

a user-defined name, which refers to a group or field type, when the character string content is not given directly. The group or field type specification is used to specify characteristics such as length and delimiters which can be used to identify the header when this is necessary. Although a GROUP or FIELD

statement may be used to describe a header, it is not a group or field in the sense that these are components of records. In this instance these statements are used in a second way which should not be confused with their primary use in record specification. This parameter is only used when describing the structure of a medium which provides labels at the lowest level.

(x) TLR: trailer. This optional parameter is used to specify control information which occurs at the tail of a basic block. It is specified and used in the same way as the header parameter, described as parameter (ix), above.

Examples of the use of the BBLOCK statement will be given in the section discussing the individual devices.

3.2 BLOCK Statements

format	BLOCK (block name [, addressing scheme]; (list), ..., (list) [; HDR: header] ... [; HDR: header] [; TLR: trailer] ... [; TLR: trailer])
parameters	(i) block name is an unindexed user-defined name. (ii) addressing scheme is a string of parameters of the form: address length, address order, base, start address where: a) address length is a string of the form n, where n is an integer; b) base is a string of the form n, where n is an integer; c) start address is a CONSTANT statement with the format described in Section 1.4.2. (iii) list is a string of parameters of the form: b/block name, occurrence, repetition number, repetition uniformity [, address level, address scope] where: a) b/block name is an unindexed user-defined name; b) occurrence is either the system name: MANDATORY (or simply M), or OPTIONAL (or simply O);

c) repetition number is either the string n, where n is an integer, or the system name NOLIM;

d) repetition uniformity is either the system name:
FIXED (or simply F), or
VARIABLE (or simply V);

e) address level is either the system name CL, or NL;

f) address scope is either the system name WOT, or WT.

(iv) header is either a CONSTANT statement with the format described in Section 1.4.2, or an unindexed user-defined name.

(v) trailer is either a CONSTANT statement with the format described in Section 1.4.2, or an unindexed user-defined name.

usage of the statement

The BLOCK statement is used to specify a type of block in the storage structure. That is, it specifies the structure and implemenation of a set of basic blocks and/or other blocks* in terms of the following parameters:

(i) block name. This parameter gives the name for blocks of the type being specified.

(ii) addressing scheme. This parameter is used to specify the addressing scheme associated with the basic blocks and/or subordinate blocks in the block being specified in terms of the following parameters:

* Blocks included in other blocks are called subordinate blocks.

a) address length. This parameter specifies the length of the bit string needed to represent the address;

b) base. This parameter gives the base of the number system in which addresses are to be given.

c) start address. This parameter gives the address to be used for the first block or basic block to which the scheme is applied.

(iii) (list), ..., (list). Each list specifies how a basic block or subordinate block is to be implemented as part of the block being specified. The order in which the basic blocks and subordinate blocks are listed gives the order in which they are to occur.

a) name. This parameter names the type of a basic block or subordinate block.

b) occurrence. This parameter specifies whether basic blocks or subordinate blocks of the type named by parameter (a) are optional or mandatory. The parameter is assigned the system name:

MANDATORY (or M), when the basic blocks or subordinate blocks of the type named must occur in each block, and

OPTIONAL (or O), when the basic blocks or subordinate blocks are not mandatory.

c) repetition number. This parameter specifies the number of basic blocks or subordinate blocks of the type named by parameter (a) are to occur in each block. The parameter is assigned the string:
n, when at most basic blocks or subordinate blocks may occur, and
NOLIM, when any number may occur.

d) repetition uniformity. This parameter specifies whether the parameter, repetition number, gives the exact or the maximum number of basic blocks or subordinate blocks. The parameter is assigned the system name:
FIXED (or F), when the repetition number gives the exact number, and
VARIABLE (or V), when it gives the maximum number.

e) address level. This parameter specifies whether the addressing scheme given by parameter (ii) is for blocks of the type named by parameter (a), or to basic blocks or subordinate blocks included in that block. The parameter is assigned the system name:

CL, when a basic block type is named by parameter (a) or a subordinate block type is named and the addressing scheme is to be applied to blocks of these type; and

NL, when the addressing scheme is to be applied to the basic blocks at the next level.

f) address scope. This parameter specifies how the basic blocks or subordinate blocks of the type named by parameter (a) are to be addressed, relative to other basic blocks or subordinate blocks in the block. The parameter is assigned the system name:

WOT, when the type named is addressed in sequence with the other types in the block, and

WT, when it is to be addressed only within the type named by parameter (a).

(iv) HDR: header. This parameter is used to specify control information which occurs at the head of a block. The statements describing the individual devices (see Section 3.4) specify whether the header is to be interpreted as a label for a track, etc. The usage of this parameter is described in Section 3.1, parameter (ix).

(v) TLR: trailer. This parameter is used to specify control information which occurs at the end of a block. Its usage is described in Section 3.1, parameter (x).

Examples of the use of the BLOCK statement will be given in the section discussing the individual devices.

3.3 POINTER Statements

format

POINTER (pointer name; pointer type; pointer mode;
pointer form)

parameters

(i) pointer name is an unindexed user-defined name.

(ii) pointer type is either the system name:
MAIN, or
DEVICE.

(iii) pointer mode is either the system name:
ABS, or the string
REL, origin
where origin is a reference name.

(iv) pointer form is either an unindexed user-defined name, or a string of the form:
field name; b/block name, ..., b/block name
where field name and b/block name are unindexed user-defined names.

usage of the statement

The POINTER statement is used to specify the implementation of a pointer in terms of the following parameters:

usage of the parameter

(i) pointer name. This parameter gives the name of the type of pointer being specified.

(ii) pointer type. This parameter is used to specify whether the pointers give the main memory addresses of records (for files to be used in main memory) or the device addresses of records. The parameter is assigned the system name:

MAIN, when the pointers are to give main memory addresses, and

DEVICE, when the pointers are to give device addresses.

(iii) pointer mode. This parameter is used to specify whether pointers give the absolute address of records, or the address relative to some record or block. The parameter is assigned the system name:

ABS, when pointers are to give the absolute address of records, and

REL, origin when pointers are to give addresses relative to some record, when the pointers give main memory addresses, or relative to some block when the pointers give device addresses.

The parameter, origin, gives the record or block type to be used as the origin.

(iv) pointer form. This parameter gives the form of a pointer in terms of the field in which it is stored, and, in the case of a pointer giving a device address, the addressing schemes for each device level used to form the pointer. The parameter is assigned:

an unindexed user-defined name, when the pointers give main memory addresses. This name identifies the field which is to contain the pointer and must appear as the first parameter in a FIELD statement;

a string of the form:

field name; block name, ..., block name

when the pointers give device addresses. The field name names the field which is to contain the pointer. The block names refer to the BLOCK statements in which the addressing schemes are specified for each block address which is to make up the total address of a record.

Example

Consider a file, stored on disk, whose records are connected by embedded pointers (called 'PTR') which give the absolute cylinder and track locations of the records. Assume the block type for cylinders is 'CYL' and the block type for tracks is 'TRK'. The following statement specifies the implementation of these pointers for the file:

```
POINTER ( 'PTR'; DEVICE; ABS; 'FIELD1'; 'CYL', 'TRK' )
```

3.4 Device Statements

Device statements are used to relate storage structure specifications to particular devices. The Device statements associate with each physical level of the device those block and basic block types which correspond to that level. The Device statements are also used to specify other medium dependent characteristics such as unit assignment and tape density for magnetic tape. There is one Device statement for each different kind of device.

3.4.1 CARD Statements

format	CARD (card specification name [, unit assignment name]; DECK: block name; CARD: bblock name, ..., bblock name)
parameters	(i) card specification name is an unindexed user-defined name. (ii) unit assignment name is a CONSTANT statement with the format described in Section 1.4.2. (iii) block name, and (iv) bblock names are unindexed user-defined names.
usage of the statement	The CARD statement is used to relate block and basic block types of a storage structure to the medium of punched cards in terms of the following parameters:
usage of the parameters	(i) card specification name. This parameter is used to refer to the relation being specified between a storage structure and the medium of cards.

(ii) unit assignment name. This parameter is used to specify the name of the device unit to be used (if selection is allowed). The unit should be identified using the naming scheme of the operating system. If no name is specified, system conventions are used to select the particular card I/O device to be used.

(iii) DECK: block name. This parameter is used to identify the block type in the storage structure which corresponds to the entire deck of cards. The block name must appear as the first parameter in a BLOCK statement. Headers and trailers specified in this BLOCK statement are interpreted as single cards.

(iv) CARD: bblock name, ..., bblock name. These parameters are used to identify those basic block types in the storage structure which correspond to single punched cards. Each bblock name must appear as the first parameter in a BBLOCK statement with the following parameters:

a) parameter (ii) is 80.

b) parameter (iii) is FIXED (or F).

c) parameters (ix) and (x) do not appear.

Example

The following statements relate a storage structure to the medium of punched cards for a file which is to contain records of a type 'X' having a fixed length of 160 characters and stored 1 record to 2 cards. There is only one type of basic-block in the storage structure and one header and trailer card. The header card contains the string: START OF DATA, starting in column 1. The trailer card contains the string: END OF DATA, starting in column 1.

```
CARD ( 'INPUT CARDS';
   DECK: 'CARD-DECK'; CARDS: 'CARD' )

BLOCK ( 'CARD-DECK'; SPEC;
   ( 'CARD', M, NOLIM, V );
   HDR: CONSTANT ( START OF DATA, ASCII );
   TLR: CONSTANT ( END OF DATA, ASCII ) )

BBLOCK ( 'CARD'; 80, FIXED; 1, 2, FIXED;
   START: 'X' )
```

3.4.2 TAPE Statements

format

```
TAPE ( tape specification name [, D: density ]
    [, unit assignment name ];
    TAPE: block name; TAPE BLOCK:
    bblock name, ..., bblock name )
```

parameters

(i) tape specification name is an unindexed user-defined name.

(ii) density, and

(iii) unit assignment name are CONSTANT statements with the format described in Section 1.4.2.

(iv) block name, and

(v) bblock names are unindexed user-defined names.

usage of the statement

The TAPE statement is used to relate block and basic block types of a stroage structure to the medium of magnetic tape in terms of the following parameters:

usage of the parameters

(i) tape specification name. This parameter is used to refer to the relation being specified between a storage structure and the medium of magnetic tape.

(ii) density. This parameter is used to specify tape density for tape units. The CONSTANT statement is used to give the density in the form required by the operating system.

(iii) unit assignment name. This parameter is used to specify the particular tape derive to be used. The unit should be specified using the CONSTANT statement according to the naming scheme of the operating system.

(iv) TAPE: block name. This parameter identifies the block type in the storage structure which corresponds to an entire magnetic tape. The block name must appear as

the first parameter in a BLOCK statement. Headers and trailers specified in this BLOCK statement are interpreted as single tape blocks.

(v) TAPE BLOCK: bblock name, ..., bblock name. These parameters are used to identify those basic block types in the storage structure which correspond to single physical tape blocks. Each bblock name must appear as the first parameter in a BBLOCK statement. Headers and trailers specified in this BBLOCK statement are interpreted as the beginning or ending storage positions in the physical tape block.

Example

The following statements relate a storage structure to a magnetic tape. The storage structure is to contain a file stored in fixed length physical blocks of 2064 bytes. Each physical block is to contain 25 fixed length records of type 'X'. The file is to be preceded by two 80 byte ASCII header labels, 'STANDARD VOLUME LABEL' and "STANDARD FILE LABEL', and a 1 byte ASCII tapemark 'TM'; and followed by a single 80 byte ASCII label 'STANDARD FILE TLR LABEL' and two 2 byte ASCII tapemarks 'TM'. Each physical block is to be preceded by a 16 bute ASCII header 'KEY'.

```
TAPE ( 'TAPE1'; TAPE: 'FILE-BLOCK';
    TAPE BLOCK: 'TAPE-BLOCK' )
```

```
BLOCK ( 'FILE-BLOCK'; SPEC;
   ( 'TAPE-BLOCK', M, NOLIM, V );
   HDR: 'STANDARD VOLUME LABEL';
   HDR: 'STANDARD FILE LABEL';
   HDR: 'TM';
   TLR: 'STANDARD FILE TLR LABEL';
   TLR: 'TM';
   TLR: 'TM' )
BBLOCK ( 'TAPE-BLOCK'; 2064, FIXED;
   25, 1, FIXED; START: 'Y';
   HDR: 'KEY' )
FIELD ( 'STANDARD VOLUME LABEL', ASCII, C, 80, F, C )
FIELD ( 'STANDARD FILE LABEL', ASCII, C, 80, F, C )
FIELD ( 'TM', ASCII, C, 1, F, C )
FIELD ( 'STANDARD FILE TLR LABEL', ASCII, C, 80, F, C )
FIELD ( 'KEY', ASCII, C, 18, F, C )
```

3.4.3 DISK Statements

format

```
DISK ( disk specification name [, unit assignment name ];
   DISK: block name; CYLINDER: cylinder name, ..., cylinder
   name; TRACK: track name, ..., track name;
   TRACK BLOCK: bblock name, ..., bblock name )
```

parameters

(i) disk specification name is an unindexed user-defined name.

(ii) unit assignment name is a CONSTANT statement with the format described in Section 1.4.2.

(iii) block name,

(iv) cylinder name,

(v) track name, and

(vi) bblock name are unindexed user-defined names.

usage of the statement

The DISK statement is used to relate block and basic block types of a storage structure to the medium of a disk in terms of the following parameters:

usage of the parameters

(i) disk specification name. This parameter is used to refer to the relation being specified between a storage structure and the medium of a disk.

(ii) unit assignment name. This parameter specifies the particular disk drive. The unit should be specified using the CONSTANT statement according to the naming scheme of the operating system.

(iii) DISK: block name. This parameter identifies the block type in the storage structure which corresponds to an entire disk. The block name must appear as the first parameter in a BLOCK statement. Header and trailer parameters are interpreted as single track blocks at the beginning and end track in the disk.

(iv) CYLINDER: cylinder name, ..., cylinder name. These parameters identify those block types in the storage structure which correspond to single cylinders. Each name must appear as the first parameter in a BLOCK statement. Header and trailer parameters are interpreted as single track blocks at the beginning and end tracks in the cylinder.

(v) TRACK: track name, ..., track name. These parameters identify those block types in the storage structure which correspond to single tracks. Each name must appear as the first parameter in a BLOCK statement. Header and trailer parameters are interpreted as single track blocks at the beginning and end of the track.

(vi) TRACK BLOCK: bblock name, ..., bblock name. These parameters are used to identify those basic block types in the storage structure which correspond to single physical track blocks. Each name must appear as the first parameter in a BBLOCK. Header and trailer parameters are interpreted as single track blocks preceding and following the track block in question.

Example

The following statements relate a storage structure to the disk medium for a file of fixed length records of type 'X'. The records are to be positioned in fixed length track blocks which are the size of a track (74,000 bits) and there are to be 90 records per track block. There is to be only one

type of track block and cylinder block. No labels are to appear.

```
DISK ( 'DISK1'; DISK: 'DISK-BLOCK';
   CYLINDER: 'CYLINDER-BLOCK';
   TRACK: 'TRACK-BLOCK';
   TRACK-BLOCK: 'TRACK-BBLOCK' )
BLOCK ( 'DISK-BLOCK'; SPEC;
   ( 'CYLINDER-BLOCK', M, NOLIM, V ) )
BLOCK ( 'CYLINDER-BLOCK', SPEC;
   ( 'TRACK-BLOCK', M, 20, V ) )
BLOCK ( 'TRACK-BLOCK', SPEC;
   ( 'TRACK-BBLOCK', M, 1, F ) )
BBLOCK ( 'TRACK-BBLOCK'; 74000, FIXED;
   90, 1, F; START: 'Z' )
```

3.4.4 TTY Statements

format

```
TTY ( teletype specification name [, unit assignment name ];
    TTY FILE: block name; PAGE: page name, ..., page name;
    LINE: line name, ..., line name )
```

parameters

(i) teletype specification name is an unindexed user-defined name.

(ii) unit assignment name is a CONSTANT statement with the format described in Section 1.4.2.

(iii) block name,

(iv) page name, and

(v) line name are unindexed user-defined names.

usage of the statement

The TTY statement is used to relate block and basic block types of a storage structure to the medium of teletypewriter input and output in terms of the following parameters:

usage of the parameters

(i) teletype specification name. This parameter is used to refer to the relation being specified between a storage structure and the teletypewriter medium.

(ii) unit assignment name. This parameter specifies the particular teletypewriter.

(iii) TTY FILE: block name. This parameter is used to identify the block type in the storage structure which corresponds to the entire input or output from the teletypewriter. The block name must appear as the first parameter in a BLOCK statement. Headers and trailers specified in this BLOCK statement are interpreted as lines.

(iv) PAGE: page name, ..., page name. These parameters are used to identify those block types in the storage structure which correspond to single pages. Each page name must appear as the first parameter in a BLOCK statement. Header and trailer parameters are interpreted as lines at the beginning and end of a page.

(v) LINE: line name, ..., line name. These parameters are used to identify those basic block types in the storage structure which correspond to single lines. Each line name must appear as the first parameter in a BBLOCK statement with the following parameters:

a) parameter (ii) is 120.

b) parameter (iii) is FIXED.

c) parameters (ix) and (x) are interpreted as columns at the beginning and end of lines.

Example

The following statements relate a storage structure to the teletypewriter medium for a file of fixed length records of type 'Q'. The records are to be positioned 1 per line and there are to be 64 lines per page. There is to be only one type of line and page and no headers or trailers.

```
TTY ( 'TTY1'; TTY FILE: 'TTY-BLOCK';
   PAGE: 'TTY-PAGE';
   LINE: 'TTY-LINE' )

BLOCK ( 'TTY-BLOCK'; SPEC;
   ( 'TTY-PAGE', M, NOLIM, V ) )
BLOCK ( 'TTY-PAGE', SPEC;
   ( 'TTY-LINE', M, 64, V ) )
BBLOCK ( 'TTY-LINE', 120, FIXED;
   1, 1, FIXED; START: 'Q' )
```

4. Conversion Specification Statements

The following terminology will be used in presenting the Conversion Specification statements:

(i) Data conversion is a process which, given a bit string representation of a file (or set of files) on a storage medium, produces the bit string representation on a (different) storage medium of a file (or set of files) whose fields contain values obtained from the fields of the first file (or set of files).

(ii) Given a set of files $X_1, \ldots, X_n$ whose data are to be converted into a set of files $Y_1, \ldots, Y_n$; we call the files $X_1, \ldots, X_n$ the source files and the files $Y_1, \ldots, Y_n$ the target files of the conversion.

(iii) The fields, groups and records of a source (target) file will be called source (target) fields, source (target) groups, and source (target) records, respectively.

(iv) An association list is a list which gives for each different type of field in the target files, the source field which is to provide its value, and any information needed to locate the source record containing the source field.

The Conversion Specification statements are used to specify the conversion of a set of source files to a set of target files in terms of descriptions of those source and target files and an association list. The Association Statements of Section 4.1 specify the association list and the CONVERT statement of Section 4.2 specifies the conversion.

4.1 Association Statements

The association list identifies for each field type in the target file, a field type in the source file which provides its value. For a given target record, values may be obtained from one or more source records and from one or more repetitions of a single field or group type in a source record.

To refer to such records, fields and groups unambiguously, reference names are modified in the following way:

The record (field, group) name is replaced by the SOURCE statement of Section 4.1.2.

Reference names modified in this way are also called reference names. When these names are used in an association list, they refer to records (fields, groups) which have already been used as the source of values for target fields.

4.1.1 ASSOCIATE Statements

format	ASSOCIATE (association name; (association entry), ..., (association entry))
parameters	(i) association name is an unindexed user-defined name. (ii) association entry is a string of parameters of the form: target name, source name [; R: criterion name] [; F/G: criterion name] where:

a) target name is a reference name.

b) source name is a reference name.

c) and d) criterion name is an unindexed user-defined name.

usage of the statement

The ASSOCIATE statement is used to specify for each field in the target files, the source field which is to provide its value in terms of the following parameters:

usage of the parameters

(i) association name. This parameter gives the name to be used in referring to the association list being specified.

(ii) (association entry), ..., (association entry)
These parameters specify for each type of target field, a source field which is to provide the value. There is one association entry for each type of target field in each target file.

a) target name. This parameter gives the type of target field. The name must be modified by strings of the form: OF record name, and OF file name to avoid ambiguities as to the record and file in which the target field is to occur.

b) source name. This parameter gives the type of source field which is to provide the value for the target field. This name must also be modified by strings of the form: OF record name,

and OF file name to avoid ambiguities as to the record and file in which it occurs.

c) R: criterion. This parameter gives the name of a criterion which is used to select the source record which contains the source field. The name must appear as the first parameter in a CRITERION statement.

d) F/G: criterion. This parameter gives the name of a criterion which is used to select a particular occurrence of a repeating field or group to use as a source field or group. The name must appear as the first parameter in a CRITERION statement.

usage conventions

The following conventions must be observed in using the ASSOCIATE statement whenever the source field is a repeating field.

1. If the target field (or parameter (ii)a) is not a repeating field, then giving the source field name unmodified by an index results in one target record being formed for each repetition of the source field.

2. If the target field is not a repeating field, then giving the source field name modified by an index results in only one target record being formed and the remaining repetitions of the source field not being used.

3. If the target field is to repeat an unlimited number of times, then giving the source field name unmodified by an index results in the target field repeating the same number of times as the source field.

4. If the target field is to repeat a fixed or bounded number of times which is less than the number of times the source field repeats, then giving the source field name unmodified results in target records being formed until all repetitions of the source field are used.

5. If the target field is to repeat a fixed or bounded number of times which is less than the number of times the source field repeats, then modifying the target name and the source name by an index and creating a separate entry for each repetition of the target field results in only one target record being formed and the remaining repetitions of the source field not being used.

Example

Consider a source record of type 'PUBLICATION' in file 'F1' and a target record of type 'PERSON' in file 'F2', satisfying the following GDDL descriptions:

```
RECORD ( 'PUBLICATION', 'PUB-GRP' )
GROUP ( 'PUB-GRP', SPEC;
   ( 'NAME', M, 1, F ),
   ( 'AGE', M, 1, F ),
   ( 'SEX', M, 1, F ),
   ( 'BOOK-GRP', 0, NOLIM, V ) )
```

```
GROUP ( 'BOOK-GRP', SPEC;
   ( 'TITLE', M, 1, F ),
   ( 'PAGES', M, 1, F ),
   ( 'DATE', M, 1, F ) )
FIELD ( 'NAME', ASCII, C, 15, F, C )
FIELD ( 'AGE', ASCII, C, 2, F, C )
FIELD ( 'SEX', ASCII, C, 1, F, C )
FIELD ( 'TITLE', ASCII, C, 20, F, C )
FIELD ( 'PAGES', ASCII, C, 4, F, C )
FIELD ( 'DATE', ASCII, C, 4, F, C )

RECORD ( 'PERSON', 'PERS-GRP' )
GROUP ( 'PERS-GRP', SPEC;
   ( 'NAME', M, 1, F ),
   ( 'AGE', M, 1, F ),
   ( 'SEX', M, 1, F ) )
FIELD ( 'NAME', EBCDIC, C, 15, F, C )
FIELD ( 'AGE', EBCDIC, C, 2, F, C )
FIELD ( 'SEX', EBCDIC, C, 1, F, C )
```

To specify the conversion of records of type 'PUBLICATION' to records of type 'PERSON', the following association list (called 'ASSOC12') is provided:

```
ASSOCIATE ( 'ASSOC12',
   ( 'NAME' OF 'PERSON' OF 'F2',
       'NAME' OF 'PUBLICATION' OF 'F1' ),
```

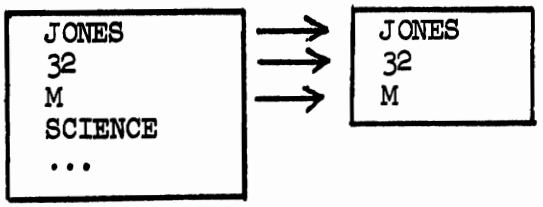
```
( 'AGE' OF 'PERSON' OF 'F2',
    'AGE' OF SOURCE ( 'NAME' OF 'PERSON' OF 'F2' ) ),
( 'SEX' OF 'PERSON' OF 'F2',
    'SEX' OF SOURCE ( 'NAME' OF 'PERSON' OF 'F2' ) ) )
```

The use of the system name, SOURCE, is explained in Section 4.1.2.

Using this association list, a record:

```
JONES
32
M
SCIENCE I
384
1958
SCIENCE II
501
1963
```

is converted as follows:



All source records of file F1 are converted in this way.

4.1.2 SOURCE Statements

format	SOURCE (target name [; criterion name])
parameters	(i) target name is a reference name. (ii) criterion name is an unindexed user-defined name.
usage of the statement	The SOURCE statement is used as part of a reference name to refer to a particular record, group or field which was the source of a value for a target field. If a reference to a particular source group or field that contains the value for a certain target field is required, then the form: source reference name - SOURCE statement is used. If, however, a reference to a source group or field that appears in a record which contains the value for a certain target field is required, then the usual form: source reference name OF SOURCE statement is used. This distinction is important when the group or field is repeating in the source record.
usage of the parameters	(i) target name. This parameter gives the type of the target field for which a value was provided. (ii) criterion name. This parameter is used when the target field repeats and it is not possible to use an index to identify the appropriate repetition. The parameter names a criterion which identifies the appropriate field. The name must appear as the first

Example

Consider a source record of the type 'PUBLICATION' described in the Example of Section 4.1.1 and a source record of the type 'BOOK' in file 'F3' satisfying the following GDDL description:

```
RECORD ( 'BOOK', 'BOOK-GRP' )
GROUP ( 'BOOK-GRP', SPEC;
   ( 'TITLE', M, 1, F ),
   ( 'AUTHOR', M, NOLIM, V ),
   ( 'PAGES', M, 1, F ),
   ( 'DATE', M, 1, F ) )
FIELD ( 'TITLE', ASCII, C, 20, F, C )
FIELD ( 'AUTHOR', ASCII, C, 15, F, C )
FIELD ( 'PAGES', ASCII, C, 4, F, C )
FIELD ( 'DATE', ASCII, C, 4, F, C )
```

To specify the conversion of records of type 'PUBLICATION' to records of type 'BOOK', the following association list (called 'ASSOC13') is specified:

```
ASSOCIATE ( 'ASSOC13',
  ( 'TITLE' OF 'BOOK' OF 'F3',
      'TITLE' OF 'PUBLICATION' OF 'F1' ),
  ( 'AUTHOR' OF 'BOOK' OF 'F3',
     'NAME' OF 'PUBLICATION' OF 'F1'; 'CRIT13' ),
  ( 'DATE' OF 'BOOK' OF 'F3',
     'DATE' OF 'BOOK-GRP' OF SOURCE ( 'TITLE' OF 'BOOK' OF
        'F3' ) ),
```

```
( 'PAGES' OF 'BOOK' OF 'F3',
    'PAGES' OF 'BOOK-GRP' OF SOURCE ( 'TITLE' OF 'BOOK'
      OF 'F3' ) ) )
```

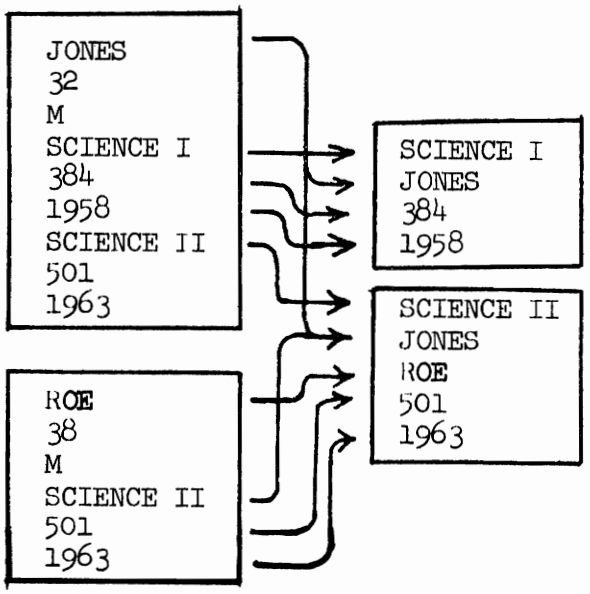
```
CRITERION ( 'CRIT13', ( 'TITLE' OF 'PUBLICATION' OF 'F3' )
  = ( 'TITLE' OF 'BOOK' OF SOURCE ( 'TITLE' ) ) )
```

Using this association list, the records:

```
JONES
32
M
SCIENCE I
384
1958
SCIENCE II
501
1963
```

```
ROE
38
M
SCIENCE II
501
1963
```

are converted as follows:



4.2 CONVERT Statements

format

```
CONVERT ( SOURCE FILES: file name, ..., file name;
          TARGET FILES: file name, ..., file name;
          association name )
```

parameters

(i) and (ii) file name is an unindexed user-defined name.

(iii) association name is an unindexed user-defined name.

usage of the statement

The CONVERT statement is used to specify that a set of source files is to be converted into a set of target files in terms of the following parameters:

usage of the parameters

(i) SOURCE FILES: file name, ..., file name.
These parameters give the names of the source files to be converted. Each name must appear as the first parameter in a FILE statement.

(ii) TARGET FILES: file name, ..., file name.
These parameters give the names of the target files to be created. Each name must appear as the first parameter in a FILE statement.

(iii) association list. This parameter gives the name of the association list which specifies for each field in the target files, the field in the source files which is to provide its value.

Example

Consider a file 'F1' consisting only of records of type 'PUBLICATION' and a file 'F2' consisting only of records of type 'PUBLICATION' and a file 'F2' consisting only of records of type 'PERSON' as described in the example of Section 4.1.1. The following statement specifies the conversion of file 'F1' into the file 'F2':

```
CONVERT ( SOURCE FILES: 'F1';
          TARGET FILES: 'F2';
          'ASSOC12' )
```

Appendix B

EXAMPLES OF GDDL DESCRIPTIONS

The examples presented in this Appendix demonstrate the descriptive power of GDDL.

Example 1 illustrates the interfacing of files with new programs and different programming languages.

Example 2 illustrates the interfacing of files with different operating systems.

The data structures selected for use in these examples are actual structures which have been implemented and are currently in use.

Example 1.

Use of GDDL to Convert Files for Interfacing with New Programs and Different Programming Languages

This example demonstrates how GDDL can be used to describe the converstion of a file of records created by a program written in an Assembly Language to a form in which the data contained in the file can be used by a COBOL program. The records to be converted are records produced by the Extended Data Management Facility (EDMF) of the University of Pennsylvania. This conversion requires the description of a set of data in a complex structure at the record level.

To perform this conversion, all that is required is:

1) a GDDL description of the file of EDMF records;
2) a GDDL description of a file which can be processed by the COBOL program;
3) a GDDL description of the relationships between values in the COBOL records and those values in the EDMF records; and
4) the file of EDMF records.

The conversion process for this example is illustrated in Figure 1-1.

The descriptions of the file of EDMF records, the COBOL file and the relationship between them are discussed separately.

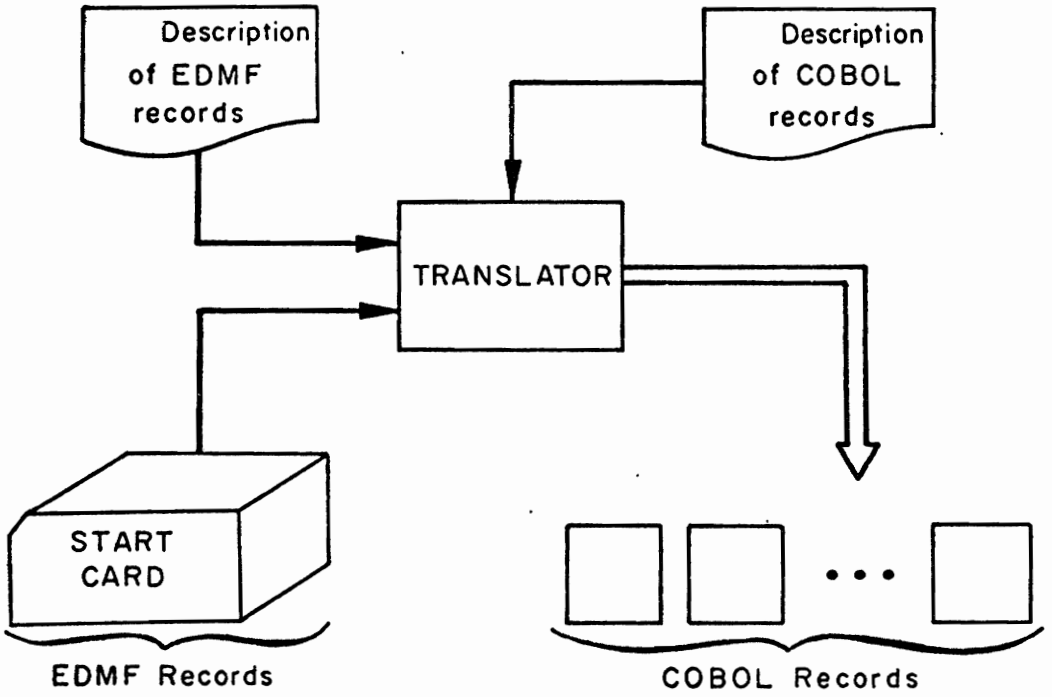


Figure 1-1 EDMF to COBOL Record Conversion

1.1 The File of EDMF Records

The EDMF provides its users with a very comprehensive record organization feature. Data in the EDMF consists of two parts: the first part is called an attribute and the second part is called a value. An EDMF record is a collection of such attribute-value pairs. These pairs can be organized into a hierarchic structure in which attributes may repeat, and occur optionally, and in which values may have fixed or variable sizes and may be interpreted as numeric or alphanumeric strings.

In this example each of the user's EDMF records is to contain data on a book. This data will include:

i) a code number for each book. The code number has the attribute CODE NUMBER. This attribute occurs exactly once in each record. The values have a fixed size of 8 bytes and are stored as an alphanumeric string.

ii) the authors' names. An author has the attribute AUTHOR. This attribute must occur one or more times in each record. The values for this attribute have variable sizes and are stored as alphanumeric strings.

An example of a collection of attribute-value pairs satisfying this description is given in Figure 1-2.

CODE NUMBER, BINER540

AUTHOR, BIVENS, R.L.

AUTHOR, METROPOLIS, N.

Figure 1-2 Example of a Collection of Attribute-Value Pairs Forming an EDMF Record

These attribute-value pairs are stored in the organization illustrated in Figure 1-3.

Storage Required	Data Stored in EDMF Record	
3 BYTES	SIZE OF RECORD	CORE FORMAT READER
5 BYTES	REFERENCE NUMBER, UNPACKED	
1 BYTE	CONTROL INFORMATION	
3 BYTES	LENGTH OF ATTR.-VALUE ENTRY	ATTRIBUTE VALUE ENTRY
1 BYTE	CONTROL INFORMATION	
1 BYTE	NUMBER OF DIRECTORY LISTS	
2 BYTES	LENGTH OF ATTRIBUTE	
VARIABLE	ATTRIBUTE	
3 BYTES	LENGTH OF VALUE	
VARIABLE	VALUE	
3 BYTES	LENGTH OF ATTR.-VALUE ENTRY	. . . . .
1 BYTE	CONTROL INFORMATION	
1 BYTE	NUMBER OF DIRECTORY LISTS	
2 BYTES	LENGTH OF ATTRIBUTE	
VARIABLE	ATTRIBUTE	
3 BYTES	LENGTH OF VALUE	
VARIABLE	VALUE	
	. . . . .	

Figure 1-3 EDMF Record

The length specified in the 3 byte "Size of Record" entry includes the 9 byte Header size.

It is assumed that the user's EDMF records, ordered sequentially by CODE NUMBER, will be input on cards. The conversion process must extract the values from these records that are to be organized for use with the COBOL program.

The GDDL description of the file of EDMF records is given below. Following the GDDL description are explanations of the GDDL statements (as they are needed).

GDDL Description of the file of EDMF records:

```
1.  FILE ( 'EDMF FILE'; 'EDMF-LK';  'EDMF-BLK';  'EDMF CARDS' )

2.  LINK ( 'EDMF-LK';  'BOOK EDMF', 'BOOK EDMF';
           'CRIT-EDMF', SEQUEN; 1, FIXED )

3.  CRITERION ( 'CRIT-EDMF', ( 'CRIT-E1' )  AND  ( 'CRIT-E2' ) )

4.  CRITERION ( 'CRIT-E1',  ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF',
          H ))  LT ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF', T )) )

5.  CRITERION ( 'CRIT-E2',  ALLOCC ( X1;  NOT (
           ( ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF', X1 ))  LT
             ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF', T ))  AND
           ( ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF', H ))   LT
             ( 'CODE NUMBER' OF OCC ( 'BOOK EDMF', X1 )) ) ) ))

6.  RECORD ( 'BOOK EDMF', 'BOOK GROUP' )

7.  GROUP ( 'BOOK GROUP', SPEC;
           ( 'HEADER', M, 1, FIXED ),
           ( 'DATA1', M, 1, FIXED ),
           ( 'DATA2', M, NOLIM, V ) )

8.  GROUP ( 'HEADER', SPEC;
           ( 'RCD SIZE', M, 1, FIXED ),
           ( 'REF NO', M, 1, FIXED ),
           ( 'CONTROL INFO', M, 1, FIXED ) )
```

```
9.    FIELD ( 'RCD SIZE', EBCDIC, C, 3, FIXED, B )
10.   FIELD ( 'REF NO', EBCDIC, C, 5, FIXED, C )
11.   FIELD ( 'CONTROL INFO' EBCDIC, C, 1, FIXED, C )

12.   GROUP ( 'DATA1', SPEC;
                ( 'A-V HEADER', M, 1, FIXED ),
                ( 'CODE ENTRY', M, 1, FIXED ) )
13.   GROUP ( 'A-V HEADER', SPEC;
                ( 'A-V LENGTH', M, 1, FIXED ),
                ( 'CONTROL INFO', M, 1, FIXED ),
                ( 'DIREC NO', M, 1, FIXED ) )

14.   FIELD ( 'A-V LENGTH;, EBCDIC, C, 3, FIXED, B )
15.   FIELD ( 'DIREC NO', EBCDIC, C, 1, FIXED, C )

16.   GROUP ( 'CODE ENTRY', SPEC;
                ( 'ATT LENGTH', M, 1, FIXED ),
                ( 'CODE NUMBER ATTRIB', M, 1, FIXED ),
                ( 'VAL LENGTH', M, 1, FIXED ),
                ( 'CODE NUMBER', M, 1, FIXED ) )

17.   FIELD ( 'ATT LENGTH', EBCDIC, C, 2, FIXED, B )
18.   FIELD ( 'CODE NUMBER ATTRIB', EBCDIC, C, 11, FIXED, C)
19.   FIELD ( 'VAL LENGTH', EBCDIC, C, 3, FIXED, C )
20.   FIELD ( 'CODE NUMBER', EBCDIC, C, 8, FIXED, C )
```

```
21.   GROUP ( 'DATA2', SPEC;
            ( 'A-V HEADER', M, 1, FIXED ),
            ( 'AUTH ENTRY',M, 1, FIXED ) )

22.   GROUP ( 'AUTH ENTRY', SPEC;
            ( 'ATT LENGTH', M, 1, FIXED ),
            ( 'AUTH ATTRIB', M, 1, FIXED ),
            ( 'VAL LENGTH', M, 1, FIXED ),
            ( 'AUTHOR', M, 1, FIXED ) )

23.   FIELD ( 'AUTH ATTRIB', EBCDIC, C, 6, FIXED, C )

24.   FIELD ( 'AUTHOR', EBCDIC, C, 'VAL LENGTH' OF 'AUTH ENTRY',
            V, C )

25.   BLOCK ( 'EDMF-BLK';  ( 'CARD', M, NOLIM, V ),
            HDR: CONSTANT ( START, EBCDIC );
            TLR: CONSTANT ( END OF DATA, EBCDIC ) )

26.   BBLOCK ( 'CARD'; 80, FIXED; NOLIM, 1, V;  SPLIT: 'BOOK
            EDMF';  START: 'BOOK EDMF' )

27.   CARD ( 'EDMF CARDS';  DECK: 'EDMF-BLK';  CARD:  'CARD' )
```

Explanations:

Statement 24. In this FIELD statement the length parameter is given indirectly. For each occurrence of the field 'AUTHOR' the length of the value is the value of the field 'VAL LENGTH' in the group 'AUTH ENTRY' in which the value of 'AUTHOR' occurs.

1.2 The COBOL Record Organization

For a COBOL program to use the data described in part 1 several modifications must be made:

(i) all variable length values must be converted to fixed length.

(ii) all groups and fields which may be repeated an unlimited number of times must be converted to groups which repeat no more than a fixed maximum number of times, where the number of times the group or field repeats is stored as a variable in the record.

(iii) all unnecessary fields may be eliminated.

Thus, each COBOL record must include the following data:

(i) a code number. The code number appears exactly once in each record. It has a fixed size of 8 bytes and is stored as an alphanumeric string.

(ii) the authors' names. There may be no more than three authors' names in each record. Each author's name has a fixed size of 20 bytes and is stored as an alphabetic string.

(iii) a counter to contain the number of authors' names.

The organization of these values is illustrated in Figure 1-4.

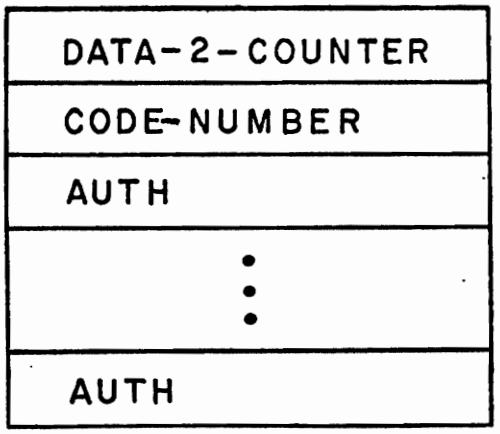


Figure 1-4 COBOL Record

The COBOL description of such a record follows:

```
01   BOOK-RECORD.
     02  DATA-2-COUNTER PICTURE IS S9999  USAGE IS COMPUTATIONAL.
     02  CODE-NUMBER PICTURE IS X(8).
     02  DATA-2 OCCURS 3 TIMES DEPENDING ON DATA-2-COUNTER.
         03  AUTHOR PICTURE IS X(20).
```

The conversion process will use the GDDL description of the EDMF record to extract the values giving the code and authors from each EDMF record. It will then use the GDDL description of the CODOL record to organize these values into a COBOL record. These records are to be ordered sequentially by CODE-NUMBER, and output on cards. The first card is to contain the string: START and the final card is to contain the string: END OF DATA.

The GDDL description of the COBOL file is given below:

GDDL Description of the COBOL file:

```
1.   FILE ( 'COBOL FILE';  'COBOL'LK';  'COBOL-BLK';
            'COBOL-CARDS' )
2.   LINK ( 'COBOL-LK';  'BOOK COBOL', 'BOOK COBOL';
            'CRIT-COBOL', SEQUEN;  1, FIXED )
3.   CRITERION ( 'CRIT-COBOL', ( 'CRIT-C1' )  AND
            ( 'CRIT-C2' ) )
4.   CRITERION ( 'CRIT-C1',  ( 'CODE-NUMBER' OF OCC ( 'BOOK
            COBOL', H ))  LT  ( 'CODE-NUMBER' OF OCC ( 'BOOK
            COBOL', T )) )
5.   CRITERION ( 'CRIT-C2',  ALLOCC ( X1;  NOT (
            (( 'CODE-NUMBER' OF OCC ( 'BOOK COBOL', X1 ))  LT
             ( 'CODE-NUMBER' OF OCC ( 'BOOK COBOL', T ))  AND
            (( 'CODE-NUMBER' OF OCC ( 'BOOK COBOL', H ))   LT
             ( 'CODE-NUMBER' OF OCC ( 'BOOK COBOL', X1 ))) ) ) )

6.   RECORD  ( 'BOOK COBOL', 'BOOK GROUP' )
7.   GROUP  ( 'BOOK GROUP', SPEC;
            ( 'DATA-2-COUNTER', M, 1, FIXED ),
            ( 'CODE-NUMBER', M, 1, FIXED ),
            ( 'DATA-2', M, 3, VARIABLE ) )

8.   FIELD ( 'DATA-2-COUNTER', EBCDIC, C, 2, FIXED, B )
9.   FIELD ( 'CODE-NUMBER', EBCDIC, C, 8, FIXED, C )
10.  GROUP ( 'DATA-2', SPEC;  ( 'AUTH', M, 1, FIXED ) )
11.  FIELD ( 'AUTH', EBCDIC, C, 20, FIXED, C )
```

```
12.     BLOCK ( 'COBOL-BLK';  ( 'CCARD', M, NOLIM, V ),
                HDR:  CONSTANT ( START, EBCDIC );
                TLR:  CONSTANT ( END OF DATA, EBCDIC ) )

13.     BBLOCK ( 'CCARD';  80, FIXED;  NOLIM, 1, V;
                SPLIT:  'BOOK COBOL';  START:  'BOOK COBOL' )

14.     CARD ( 'COBOL-CARDS';  DECK:  'COBOL-BLK';  CARD:  'CARDS' )
```

1.3 The Relationship Description

To convert records from the EDMF record format to the COBOL record format, the user must also specify where each value for the COBOL records is to be found in the EDMF records. This is specified in the single GDDL statement given below:

```
ASSOCIATE ( 'ASSOC LIST';
    ( 'CODE-NUMBER' OF 'BOOK COBOL' OF 'COBOL FILE',
        'CODE NUMBER' OF 'BOOK EDMF' OF 'EDMF FILE' ),
    ( 'AUTH' OF 'DATA-2'(1) OF 'BOOK COBOL' OF 'COBOL FILE',
        'AUTHOR' OF 'DATA2'(1) OF SOURCE ( 'CODE-NUMBER' )),
    ( 'AUTH' OF 'DATA-2'(2) OF 'BOOK COBOL' OF 'COBOL FILE',
        'AUTHOR' OF 'DATA2'(2) OF SOURCE ( 'CODE NUMBER' )),
    ( 'AUTH' OF 'DATA-2'(3) OF 'BOOK COBOL' OF 'COBOL FILE',
        'AUTHOR' OF 'DATA2'(3) OF SOURCE ( 'CODE NUMBER' )),
    ( 'DATA-2-COUNTER' OF 'BOOK COBOL' OF 'COBOL FILE',
        COUNT ( 'DATA-2' OF 'BOOK COBOL' )) )
```

Given the values of an EDMF record organization as described in section 1.1, this GDDL statement specifies that:

(i) the value of the field 'CODE-NUMBER' in the COBOL record comes from the field 'CODE NUMBER' in the EDMF record.

(ii) the values of 'AUTH' in the repeating group 'DATA-2' of the COBOL record come from the values of 'AUTHOR' in the repeating group 'DATA2' in the EDMF record. If there are more than three values of 'AUTHOR' in the EDMF record, these are not used.

(iii) the value of 'DATA-2-COUNTER' is the number of values for 'AUTH' in the COBOL record being created.

These relationships are illustrated in Figure 1-5.

1.4 The Complete Description of the Conversion

The GDDL statements given in Parts 1.1, 1.2, and 1.3 of this example, together with the GDDL statement given below, complete the description requirements for the conversion of the file of EDMF records to a form that can be used by a COBOL program.

```
CONVERT ( SOURCE FILES: 'EDMF FILE';  TARGET FILES: 'COBOL FILE';
                'ASSOC LIST' )
```

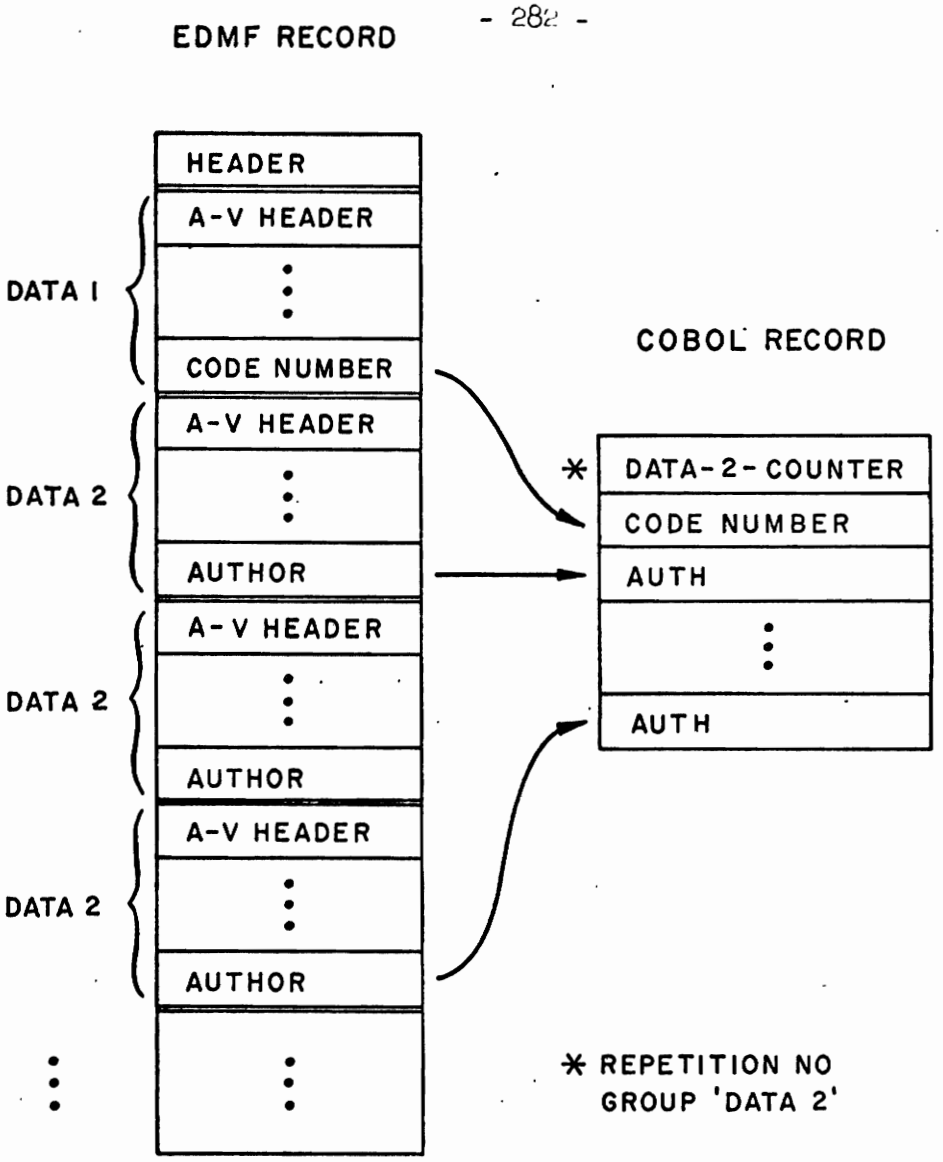


Figure 1-5 Relationships between fields in the COBOL record and fields in the EDMF record

Example 2.

Use of GDDL to Convert Files for Interfacing with Different Operating Systems

This example demonstrates how GDDL can be used to describe the conversion of a file created under one operating system to a form in which it can be accessed under a different operating system. The data to be converted is in the form of fixed length records stored on tape by the Sequential Access Method (SAM) of the RCA SPECTRA 70/46 Time Sharing Operating System (TSOS). This file is to be converted into a file on disk that can be accessed by the Indexed Sequential Access Method (ISAM) of the RCA SPECTRA 70/46 Tape Disk Operating Sytem (TDOS). This conversion requires the description of sets of data in complex structures at the file and storage levels.

To perform this conversion, the following descriptions and data are required:

1) a GDDL description of the structure of the SAM file;
2) a GDDL description of the structure of the ISAM file;
3) a GDDL description of the relationships between data items in the SAM file and those data items in the ISAM file; and
4) the SAM file.

The conversion process if illustrated in Figure 2-1.

The descriptions of the SAM file, the ISAM file and the relationship descriptions are discussed separately.

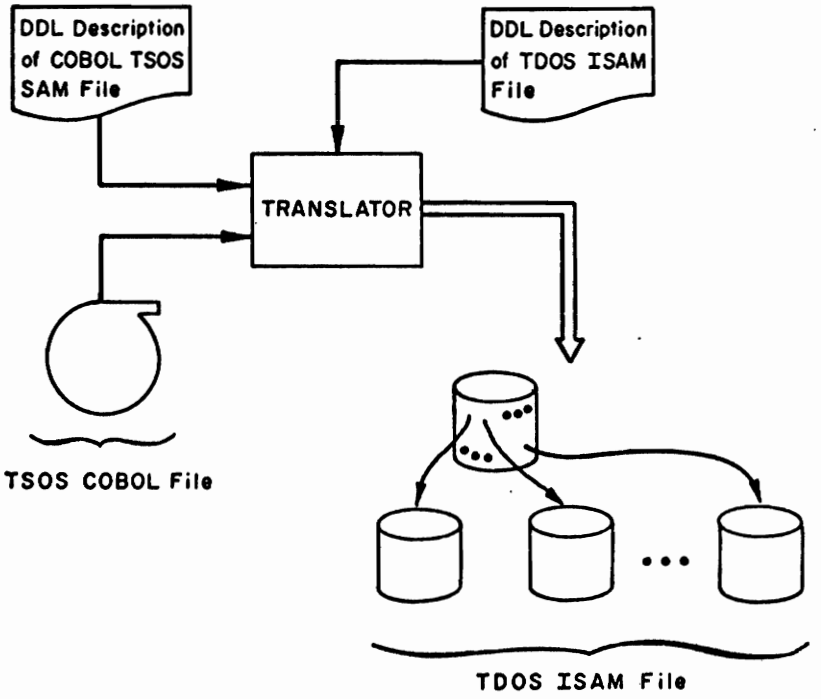


Figure 2-1 TSOS SAM File to TDOS ISAM File Conversion

2.1 The TSOS SAM File Organization

The TSOS SAM File of this example is a file produced by a COBOL program. The file contains information on bills and is assigned the name BILL-FILE. Each record, called BILL-RECORD, is of fixed length with the organization shown in Figure 2-2.

VENDOR-NAME	30 bytes
BILL-NO	10 bytes
PART-NO	10 bytes
SERIAL-NO	10 bytes
AMT	6 bytes
COST	6 bytes
DATE	6 bytes
DEPT	2 bytes
	80 bytes

Figure 2-2 BILL RECORD

The records are stored on tape. They are ordered sequentially by SERIAL-NO. The COBOL description of this file follows:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BILL-FILE
      ASSIGN TO SYS012 UTILITY
      ACCESS IS SEQUENTIAL
DATA DIVISION.
```

```
FILE SECTION.

    FD BILL-FILE

      RECORDING MODE IS F

      LABEL RECORD IS STANDARD

      DATA RECORD IS BILL-RECORD

01  BILL-RECORD.

    02  VENDOR-NAME PICTURE IS X(30).

    02  BILL-NO PICTURE IS X(10).

    02  PART-NO PICTURE IS X(10).

    02  SERIAL-NO PICTURE IS X(10).

    02  AMT PICTURE IS X(6).

    02  COST PICTURE IS X(6).

    02  DATE PICTURE IS X(6).

    02  DEPT-NO PICTURE IS X(2).
```

The file was produced using TSOS SAM, which produces a tape with the organization illustrated in Figure 2-3.

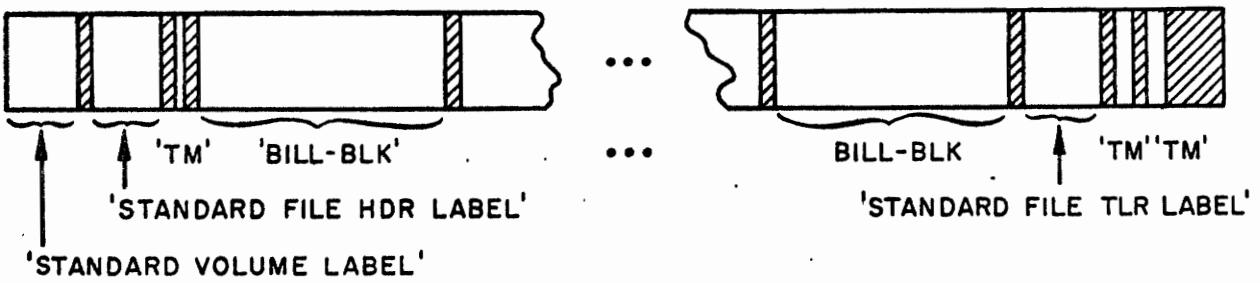


Figure 2-3 Storage Image of BILL-FILE

Each of the labels contains 80 bytes. Each physical block, 'BILL-BLK', on the tape has the organization illustrated in Figure 2-4.

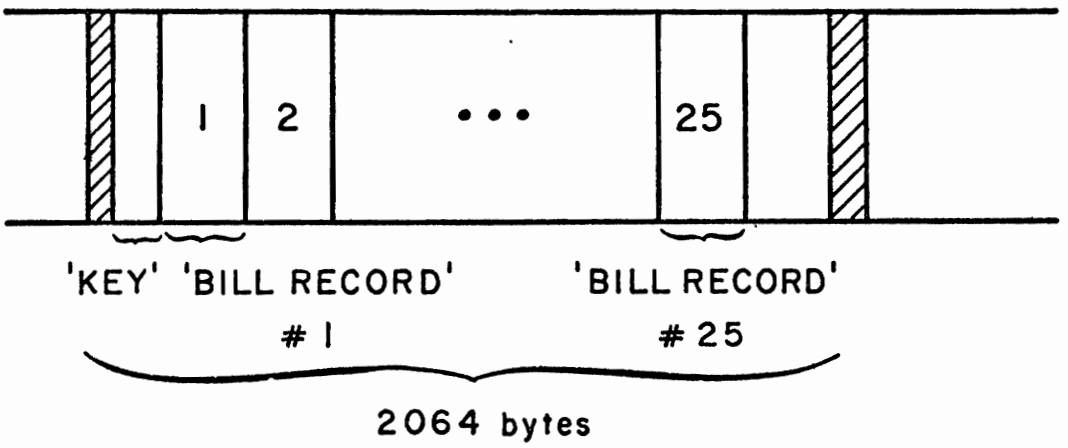


Figure 2-4 Storage Structure of 'BILL-BLK'

Each 'BILL-BLK' contains 25 'BILL-RECORD' record occurrences.

The GDDL description of the TSOS SAM file is given below. For brevity, the description of the records are omitted.

GDDL Description of the TSOS SAM file BILL-FILE:

```
1.   FILE ( 'BILL-FILE';  'BILL-LINK'; 'BILL-TAPE';  'TAPE' )
2.   LINK ( 'BILL-LINK';  'BILL RECORD', 'BILL RECORD';
            'BILL-CRIT', SEQUEN;  1, FIXED )
3.   CRITERION ( 'BILL-CRIT',  ( 'CRIT-B1' )  AND  ( 'CRIT-B2' ))
4.   CRITERION ( 'CRIT-B1',  ( 'SERIAL-NO' OF OCC ( 'BILL-
            RECORD', H ))  LT  ( 'SERIAL-NO' OF OCC ( 'BILL-
            RECORD', T )) )
5.   CRITERION ( 'CRIT-B2',  ALLOCC ( X1;  NOT (
            (( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', X1 ))  LT
             ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', T ))  AND
```

```
(( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', H ))   LT
 ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', X1 ))) ) ) )
```

6. The GDDL statements describing the record BILL-RECORD appear here.

```
7.  BLOCK ( 'BILL-TAPE';  ('BILL-BLK', M, NOLIM, V ),
          HDR:  'STANDARD VOLUME LABEL';
          HDR:  'STANDARD FILE HDR LABEL';
          HDR:  'TM';
          TLR:  'STANDARD FILE TLR LABEL';
          TLR:  'TM';
          TLR:  'TM' )
8.  FIELD ( 'STANDARD VOLUME LABEL', EBCDIC, C, 80, FIXED, C )
9.  FIELD ( 'STANDARD FILE HDR LABEL', EBCDIC, C, 80, FIXED, C )
10. FIELD ( 'TM', B, B, 8, FIXED, C )
11. FIELD ( 'STANDARD FILE TLR LABEL', EBCDIC, C, 80, FIXED, C )
12. BBLOCK ( 'BILL-BLK'; 2064, FIXED; 80, 1, FIXED;
          START: 'BILL-RECORD';  HDR: 'KEY' )
13. FIELD ( 'KEY', EBCDIC, C, 2, FIXED, C )
14. TAPE ( 'TAPE';  TAPE: 'BILL-TAPE';  TAPE BLOCK:  'BILL-BLK' )
```

2.2 The TDOS ISAM File Organization

For the file described in part 2.1 to be accessible to TDOS ISAM two modifications must be made:

(i) the records must be stored on disk under TDOS ISAM conventions; and

(ii) hierarchic indices must be created.

The emphasis of this part of the example will be on describing the overall structure of these indices rather than on the details of their implementation. Therefore, we will not discuss the addressing scheme or the implementation of the pointers stored in the indices.

The new file will be assigned the name BILL-FL to conform with TDOS naming conventions. BILL-FL will be organized as follows:

(i) the records will be ordered sequentially according to the values of the field SERIAL-NO.

(ii) the records will be blocked. Each disk block will be the size of a disk track. To simplify the example, it will be assumed that no overflow tracks are to be reserved. 19 tracks per cylinder will be used to store records.

(iii) three levels of indices will be created:

1) a track index for each data cylinder. One track per cylinder of records will be reserved for an index. Each record in this index will point to a track block. These records are not blocked.

2) an index cylinder. To simplify this example, it will be assumed that only one index cylinder is needed.

Each record in this index will point to a track index of a data cylinder. Records will be blocked. Each disk block will be the size of a disk track.

3) a track index for the index cylinder. One track of the index cylinder is reserved for an index. Each entry in this index will point to a disk block. Records are not blocked.

The organization of BILL-FL is illustrated in Figure 2-5.

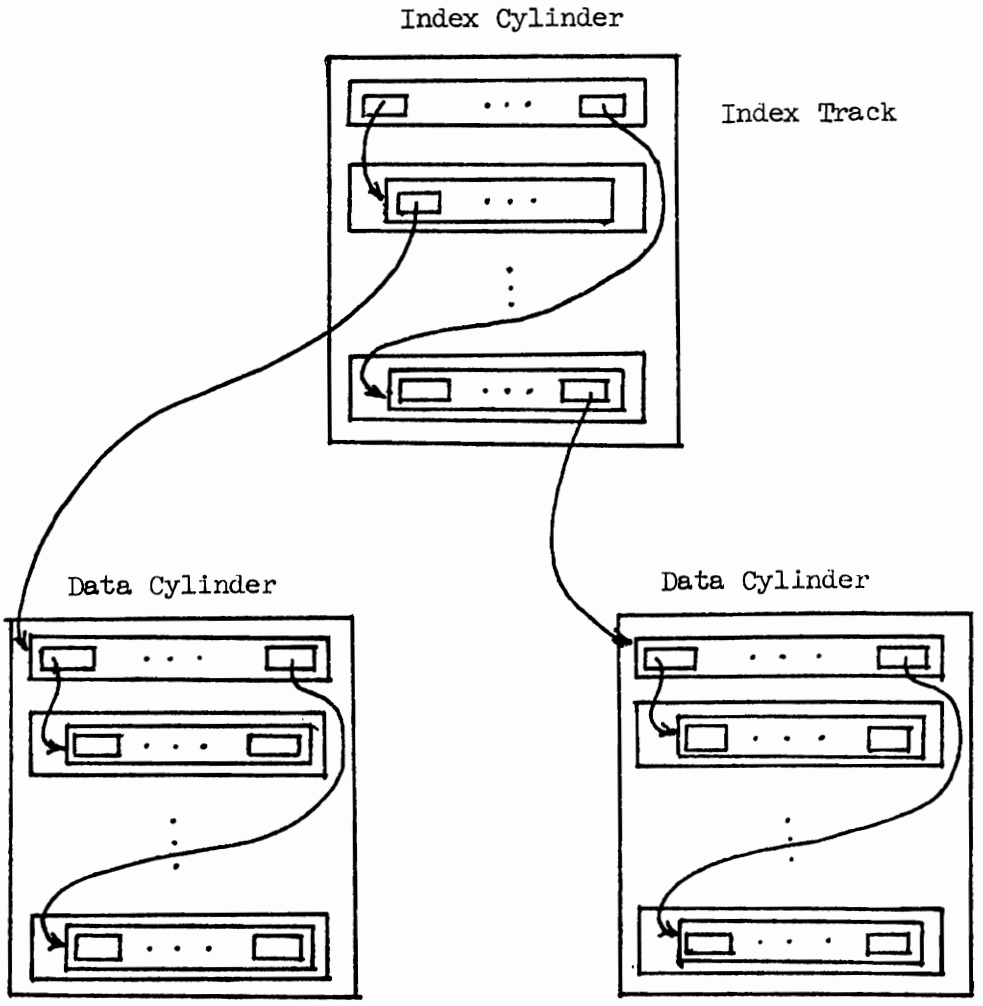


Figure 2-5. BILL-FL Structure

The RCA TDOS ISAM description of this file follows:

```
BILL-FL     DTFIS     BLKSIZE=7241,
                      KEYLEN=10,
                      KEYLOC=51,
                      PRIDEVT=DISK90,
                      PRIDEX=0,
                      RECFORM=FIXED,
                      RECSIZE=80,
                      UPINDEX=DISK90
```

The conversion process will use the GDDL description of the TSOS SAM file, BILL-FILE, to extract the values from the tape. It will then use the GDDL description given below to organize these values into the TDOS ISAM file format.

The GDDL description of the file BILL-FL is given below For brevity, the description of the records and device labels are omitted.

GDDL Description of TDOS ISAM File BILL-FL:

```
1.   FILE ( 'BILL-FL'; 'BILLS-SEQ1', 'BILLS-SEQ2'; 'FILE BLOCK';
                                                        'DISK' )
2.   LINK ( 'BILLS-SEQ1'; 'BILL-RECORD', 'BILL-RECORD';
            'BILLS-SEQ-CRIT', SEQUEN; 1, FIXED )
3.   LINK ( 'BILLS-SEQ2'; 'BILL-RECORD', 'BILL-RECORD';
            'BILLS-SEQ-CRIT', DIREC, 'BX-PTR', 'BINDX';
            1, FIXED )
4.   CRITERION ( 'BILLS-SEQ-CRIT', ( 'BSQ1' ) AND ( 'BSQ2' ) )
5.   CRITERION ( 'BSQ1', ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD',
            H )) LT ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', T )) )
```

```
6.    CRITERION ( 'BSQ2',  ALLOCC ( X1;  NOT  (
            (( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', X1 ))  LT
             ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', T ))  AND
            (( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', S ))  LT
             ( 'SERIAL-NO' OF OCC ( 'BILL-RECORD', X1 ))) ) ) )

7.          The GDDL statements describing the record
                BILL-RECORD appear here.

8.    FILE ( 'BINDX';  'BX-SEQ1', 'BX-SEQ2';  'FILE-BLOCK'; 'DISK' )

9.    LINK ( 'BX-SEQ1;  'BX-RECORD', 'BX-RECORD';
            'BILLS-SEQ-CRIT', SEQUEN;  1, FIXED )

10.   LINK ( 'BX-SEQ2;  'BX-RECORD', 'BX-RECORD';
            'BILLS-SEQ-CRIT', DIREC, 'BXX-PTR', 'BINDXX';
            1, FIXED )

11.         The GDDL statements describing the record
                BX-RECORD appear here.  These records
                form the index for each data cylinder.

12.   FILE ( 'BINDXX';  'BXX-SEQ1', 'BXX-SEQ2;  'FILE-BLOCK'; 'DISK' )

13.   LINK ( 'BXX-SEQ1'; 'BXX-RECORD', 'BXX-RECORD';
            'BILLS-SEQ-CRIT', SEQUEN;  1, FIXED )

14.   LINK ( 'BXX-SEQ2';  'BXX-RECORD', 'BXX-RECORD';
            'BILLS-SEQ-CRIT', DIREC, 'BXXX-PTR', 'BINDXXX';
            1, FIXED )

15.         The GDDL statements describing the record
                BXX-RECORD appear here.  These records
                form the index on the index cylinder.
```

```
16.   FILE ( 'BINDXXX';  'BXXX-SEQ1';  'INDEX-CYL'; 'DISK' )
17.   LINK ( 'BXXX-SEQ1';  'BXXX-RECORD', 'BXXX-RECORD';
             'BILLS-SEQ-CRIT', SEQUEN;  1, FIXED )
18.        The GDDL statements describing the record
               BXXX-RECORD appear here.  These records
               from the index cylinder.
19.   BLOCK ( 'FILE-BLOCK';
             ( 'INDEX-CYL', 0, 1, V ),
             ( 'DATA-CYL', M, NOLIM, V ) )
20.   BLOCK ( 'INDEX-CYL';
             ( 'I-INDEX-TRACK', M, 1, FIXED ),
             ( 'I-DATA-TRACK', M, 19, VARIABLE ) )
21.   BLOCK ( 'I-INDEX-TRACK';  ( 'I-INDEX-BLOCK', M, NOLIM, V ) )
22.   BBLOCK ( 'I-INDEX-BLOCK;  NOLIM, V;  1, 1, FIXED;
             START:  'BXXX-RECORD' )
23.   BLOCK ( 'I-DATA-TRACK';  ( 'I-DATA-BLOCK', M, 1, FIXED ) )
24.   BBLOCK ( 'I-DATA-BLOCK';  74,000, FIXED;  NOLIM, 1,  V;
                    START:  'BXX-RECORD' )
25.   BLOCK ( 'DATA-CYL';
             ( 'D-INDEX-TRACK', M, 1, FIXED ),
             ( 'D-DATA-TRACK', M, 19, VARIABLE ) )
26.   BLOCK ( 'D-INDEX-TRACK';  ( 'D-INDEX-BLOCK', M, NOLIM, V ) )
27.   BBLOCK ( 'D-INDEX-BLOCK';  NOLIM, V;  1, 1, FIXED;
              START:  'BX-RECORD' )
```

```
28.   BLOCK ( 'D-DATA-TRACK';  ( 'D-DATA-BLOCK', M, 1, FIXED ) )
29.   BBLOCK ( 'D-DATA-BLOCK';  74,000, FIXED;  NOLIM, 1, V;
                        START:  'BILL-RECORD' )
30.   DISK ( 'DISK';  DISK:  'FILE-BLOCK';
               CYLINDER:  'INDEX-CYL', 'DATA-CYL';
               TRACK:  'I-INDEX-TRACK', 'I-DATA-TRACK',
                       'D-INDEX-TRACK', 'D-DATA-TRACK';
               TRACK BLOCK:  'I-INDEX-BLOCK', 'I-DATA-BLOCK',
                       'D-INDEX-BLOCK', 'D-DATA-BLOCK' )
31.            The GDDL statements for describing the pointers
                   appear here.
```

Explanations:

Statements 1 - 7. These statements describe the organization of records of type BILL-RECORD in BILL-FL.

Statements 8 - 11. These statements describe the track index for each data cylinder.

Statements 12 - 15. These statements describe the index cylinder.

Statements 16 - 18. These statements describe the track index for the index cylinder.

Statements 19 - 31. These statements describe the storage structure and its implementation for BILL-FL.

2.3 The Relationship Description

To convert data from the TSOS SAM file organization to the TDOS ISAM file organization, the user must specify where each value for the ISAM records are to be found in the SAM records. This is specified in the single GDDL statement given below:

```
ASSOCIATE ( 'BILL-ASSOC';
     ( 'BILL-RECORD' OF 'BILL-FILE', 'BILL-RECORD' OF 'BILL-FL' ))
```

Given the values of a SAM record, this GDDL statement specifies that the record is to become an ISAM record without requiring any conversion.

2.4 The Complete Description of the Conversion

The GDDL statements given in Parts 2.1, 2.2, and 2.3 of this example, together with the GDDL statement given below, complete the description requirements for the conversion of the TSOS SAM file, 'BILL-FILE', into the TDOS ISAM file, 'BILL-FL'.

```
CONVERT ( SOURCE FILES: 'BILL-FILE';  TARGET FILES: 'BILL-FL';
                                      'BILL-ASSOC' )
```

Appendix C

RELATIONSHIP OF GDDL TO COBOL

In this Appendix, we show that record level GDDL is complete with respect to COBOL and more general than COBOL. We show completeness by demonstrating that the GDDL can describe all record level options provided by COBOL and we show generality by giving a set of examples of characteristics describable in GDDL but not provided by COBOL. We give these demonstrations in terms of RCA SPECTRA 70/46 COBOL.

1. The Containment of the COBOL Record Description Features in GDDL

In this section we demonstrate that GDDL can completely describe the set of data structure options provided by each COBOL clause in the record description part of the COBOL Data Division. This is done by giving a procedure for converting any such COBOL clause into its GDDL equivalent. We give below a complete list of such COBOL clauses. We then present the statement (or statement part) in GDDL which corresponds to each COBOL clause.

1.1 The COBOL Record Description Clauses

1. level-number $\left\{\begin{matrix}\underline{\text{FILLER}} \\ \text{data-name-1}\end{matrix}\right\}$
2. [REDEFINES data-name-2]
3. [PICTURE IS picture string]
4. [BLANK WHEN ZERO]
5. [OCCURS integer TIME ...]

6. [USAGE IS { DISPLAY / COMPUTATIONAL / COMPUTATIONAL-1 / COMPUTATIONAL-2 / COMPUTATIONAL-3 / INDEX }]

7. [JUSTIFIED RIGHT]

8. [VALUE IS literal]

1.2 The COBOL to GDDL Translation Procedure

Each COBOL clause and the procedure for translating it to a GDDL statement are now given:

1. level-number { FILLER / data-name-1 }

In COBOL, the level-number is used to describe a hierarchic organization for records. Every clause of types 2 - 8 describes features of a group or field which is to be referred to by the name data-name-1 or FILLER. FILLER is used when the group or field does not have a unique name associated with it. Level-number 01 is used to describe features of the entire record.

The order in which level-number clauses occur relative to other level-number clauses is important. If such a clause is followed by one with a higher level-number, then the first clause must describe a group containing the field or group named in the second clause. If a level-number clause is followed by a clause with a higher level-number, then the former must describe a field.

This clause is described in GDDL by the following statements:

a) When the clause describes a field, the following GDDL statement is used -

FIELD (data-name-1, ...)

A unique name is created when FILLER occurs in the clause.

b) When the clause describes a group, the following GDDL statement is used -

GROUP (data-name-1, SPEC;
(data-name-11, ...),
...
(data-name-1n, ...) ...)

There is an entry of the form (data-name-1i, ...) for each group or field with the name data-name-1i at level 02.

2. [REDEFINES data-name-2]

This clause is used in COBOL to specify that a different group or field of the same length may occur in the record in the place of the one being specified. This group or field must also be described in a level-number clause, and, therefore, by GDDL statements of the type described for clause 1 above. In addition, the following parameters must be used in the GROUP statement containing entries for both data-name-1 and data-name-2:

a) In the list parameter for both data-name-1 and data-name-2, the occurrence parameter must be specified as OPTIONAL (or 0).

b) In the GROUP statement, a criterion is defined for each group or field which has been redefined. The criterion states that the data can be present in the record in only one of the forms specified. This is specified in the following GDDL statements -

CRITERION (crit-name, (COUNT (data-name-1)) NE
(COUNT (data-name-2)))

3. [PICTURE IS picture string]

where picture string is a string of type:

a) alpha-form,

b) an-form,

c) numeric-form,

d) report-form, or

e) fp-form.

This clause is used in COBOL to specify data type and length for a field. Each picture string form will be discussed separately.

a) alpha-form. A string of n A's is an alpha form. It represents an EBCDIC character string of length n consisting of the alphabetic characters and the space character. This is specified in GDDL as:

FIELD (data-name-1, 'EALPHA', C, n, F, ...)

where: the 3rd, 4th and 5th parameters specify the length as being n characters and fixed, and

the 2nd parameter specifies that the character set is specified by a CHAR statement and called 'EALPHA'; it is defined as the set of EBCDIC alphabetic and space characters.

b) an-form (alpha-numeric). A string of n X's is an an-form. It represents any EBCDIC character string of length n. This is specified in GDDL as:

FIELD (data-name-1, EBCDIC, C, n, F, C ...)

where: the 3rd, 4th and 5th parameters specify the length as being n characters and fixed, and the 2nd parameter specifies that the character set is the EBCDIC character set.

c) numeric-form. A string of 9's containing the additional characters V, P, and S is a numeric form. It represents a binary or decimal, signed, fixed-point number.

When the number is binary (see USAGE IS clause), it has a length of 2 bytes when up to 4 nines appear in the numeric form, 4 bytes when up to 9 nines appear, and 8 bytes when up to 18 bytes appear. This is specified in GDDL as:

FIELD (data-name-1, EBCDIC, C, $\left\{\begin{matrix}2\\4\\8\end{matrix}\right\}$, F, N (B, R, FX) ...)

When no USAGE clause is specified for a numeric-form field, it contains a decimal number. The number has a length of n bytes when n-1 9's appear in the numeric-form. This is specified in GDDL as:

FIELD (data-name-1, EBCDIC, C, n, F, N (D, C, FX) ...)

d) report-form. The report form is a combination of the numeric-form and fp-form used for outputting data to the printer. Therefore, it will not be discussed separately.

e) fp-form (floating point). A string consisting of the characters +, -, 9, V, and E is an fp-form. It represents a decimal floating point number with a two digit exponent. This is specified in GDDL as:

FIELD (data-name-1, EBCDIC, C, n, F, N (D, C, FL:
M-data-name-1, E-data-name-1) ...)

where the 6th parameter specifies that the field is a floating point decimal number with character sign. The mantissa and exponent are described by FIELD statements of the form:

FIELD (M-data-name-1, EBCDIC, C, m, F, N (D, C, FX)
...) and

FIELD (E-data-name-1, EBCDIC, C, 2, F, N (D, C, FX)
...) CONCODE (CONSTANT (E, EBCDIC), PRX))

where m is the number of characters in the mantissa.

4. [BLANK WHEN ZERO]

This clause specifies a value to be placed in a field during computation, and does not describe the record structure. Therefore, it is not specified in the GDDL description of a COBOL record.

5. [OCCURS [integer-1] integer-2 TIMES
[DEPENDING ON data-name-3]
[{ASCENDING / DESCENDING} KEY IS data-name-2 [, data-name-3] ...]
INDEXED BY index-name-1]

This clause specifies that an attribute repeats in a record. We will discuss each subclause of this clause separately.

a) [integer-1 TO] integer-2 TIMES

When [integer-1 TO] does not appear, this clause specifies either the number of times or the maximum number of times a group or field type repeats. This is specified in GDDL as follows:

```
GROUP ( ...
          ...
    ( data-name-1, {M|O}, integer-2 {F|V} ... ),
          ... )
```

where the fourth parameter is determined by the appearance of the DEPENDING subclause.

When [integer-1 TO] does appear, this subclause specifies the maximum number of times the group or field type repeats. Integer-1 specifies a minimum number of times it must repeat. This is specified in GDDL as follows:

The group or field in question is described as two groups or fields - one of which occurs a fixed number of times (integer-1 - 1 times) and the second which occurs a maximum number of times equal to integer-2 - integer-1 + 1. The GDDL statements that specify this are:

```
GROUP ( ...
          ...
    ( data-name-1.1, M, integer-1 - 1, F ... ),
    ( data-name-1.2, M, integer-2 - integer-1 + 1, V ... ),
          ... )
```

b) [DEPENDING ON data-name-3]

This clause is used when a group or field may repeat a variable number of times. The actual repetition number is stored as a value in the field named data-name-3. This is specified in GDDL as follows:

```
GROUP ( ...
        ...
    ( data-name-1, {M|O} , data-name-3, V ... ),
        ... )
```

c) [{ASCENDING|DESCENDING} KEY IS data-name-2 [, data-name-3] ...]

This subclause is used to specify that the values of a repeating group or field are ordered in ascending or descending order by the fields named data-name-2, data-name-3, etc.

When a field repeats, data-name-2 must equal data-name-1. This is specified in GDDL as follows:

```
GROUP ( ...
        ...
    ( data-name-1, ..., {ASCEND|DESCND} ... ),
        ... )
```

When data-name-1 refers to a group which repeats, this is specified in GDDL as follows:

```
GROUP ( ...
        ...
    ( data-name-1, ... ; 0, crit-name ... ),
        ... )
```

where crit-name refers to a criterion which determines when one group is to be placed after another group. This criterion is specified as follows:

Assuming that keys: data-name-2, ..., data-name-n are given in the COBOL description, the following CRITERION statement must be specified.

```
CRITERION ( crit-name, (crit2)  OR  ...  OR  (critn) )
```

where criti, for $2 \leq i \leq n$ is specified as:

```
CRITERION ( criti, ( (data-name-2 OF OCC (data-name-1, H))
     EQ (data-name-2 OF OCC (data-name-1, T)) )  AND  ...
     AND ( (data-name-i-1 OF OCC (data-name-1, H))  EQ
       (data-name-i-1 OF OCC (data-name-1, T)) )  AND
     ( (data-name-i OF OCC (data-name-1, H))  LT
       (data-name-i OF OCC (data-name-1, T )) )  AND
       ( ALLOCC ( X1;  NOT (  ( (data-name-i OF OCC
       (data-name-1, X1))  LT  (data-name-i OF OCC
       (data-name-1, T )) )  AND ( (data-name-i OF OCC
       (data-name-1, H ))  LT  (data-name-i OF OCC
       (data-name-1, X1)) ) ) ) )
```

d) INDEXED BY index-name-1

This subclause is used to specify working storage internal to the program and does not describe data stored in the COBOL record. Therefore, it is not described in GDDL specification of the COBOL record.

6. [USAGE IS { DISPLAY / COMPUTATIONAL / COMPUTATIONAL-1 / COMPUTATIONAL-2 / COMPUTATIONAL-3 / INDEX }]

This clause is used to specify data type, and length. It can be specified at field or group level. When it is specified at group level it specifies data type and length for all of the fields in the group.

a) When usage is DISPLAY data type is character string and length is specified in a PICTURE clause.

b) When usage is COMPUTATIONAL data type is binary number and length is specified in a PICTURE clause.

c) When usage is COMPUTATIONAL-1 data type is binary, floating point number 4 bytes in length. The mantissa has a length of 24 bits. The procedure for specifying such a field is described in the discussion of fp-forms under the PICTURE clause.

d) When usage is COMPUTATIONAL-2 the field is the same as a COMPUTATIONAL-1 field except that the length is 8 bytes.

e) When usage is COMPUTATIONAL-3 data type is decimal number specified by a PICTURE clause where the character set is packed decimal (4 bits per digit). The character code is specified in GDDL in the CHAR statement.

f) Usage is INDEXED is used to describe working storage for the program and is therefore not specified in the GDDL description of a COBOL record.

7. [JUSTIFIED RIGHT]

Normally COBOL character strings are left justified with trailing blanks. This clause is used to specify that the character string in question is to be right justified with leading blanks. This is specified in GDDL as -

FIELD (data-name-1, ... ; V, R, CONSTANT (, 'E') ...)

8. [VALUE IS literal]

This clause is used to specify that a field is set to a new value during program execution. It does not affect the stored data, and thus is not specified in the GDDL description of the COBOL record.

2. The Proper Containment of the COBOL Record Description Features in GDDL

We have shown that the GDDL includes every COBOL record description feature. Now, we show that there are record level features which are describable in GDDL but not provided by COBOL. Three examples are given, each one highlighting a different feature.

Example 1. The Specification of a Characteristic in Terms of Other Characteristics

In GDDL, characteristics of records, groups and fields can be specified in terms of other characteristics. For example a user can describe in GDDL the length in bits of a field X as equal to the number of times another field Y repeats. This is specified as follows:

```
FIELD ('X', B, B, COUNT ( 'Y' ), V, ... )
```

The fourth parameter specifies that the length of field X is equal to the number of repetitions of Y.

COBOL provides no way to make characteristics dependent on other characteristics.

Example 2. Variable Characteristics and the Use of Delimiters

In GDDL, record level characteristics such as field length and repetition number can be described which vary depending on the data. In such cases, delimiters are used to indicate the beginning and/or end of the fields in question. For example, a user can specify that the length of some field X is to vary with the character $ used as a delimiter. This is described as follows:

```
FIELD ('X', B, B, NOLIM, V, ...
  CONCODE ( CONSTANT ( $, EBCDIC), PRX ),
  CONCODE ( CONSTANT ( $, EBCDIC), PTX ) )
```

The final two parameters specify that $ is to be used as a beginning and end delimiter for values of the field X.

COBOL provides no delimiter feature.

Example 3. The Specification of Bit Fields

In GDDL, field lengths can be specified in terms of any character code or in bits. Thus, a user can describe the length of a field X to be a single bit. This is specified as follows:

```
FIELD (X, B, B, 1, F, ... )
```

The third parameter specifies that length is given in bits and the fourth parameter specifies that the length is 1.

COBOL allows length to be specified in characters only.

3. Conclusion

In section 1 we show that GDDL includes every COBOL record description features. In the section 2 we saw that COBOL does not include all of the GDDL record description features. Thus, we may conclude that record level GDDL properly contains the COBOL record description features.