the resulting set of assertions would have been the same but the subset of assertions added by a particular query might have been different.

Figure 16 contains a data structure diagram for the design that was created by the system. The DOCTOR-PATIENT confluency is detected from ABOVE assertions made in the second and third queries, and the base for the confluent hierarchy, the TREATMENT record, is discovered from assertions made in the first and second queries. The ABOVE assertions from queries five and six are erased because of redundancy. Only five ABOVE assertions remain, resulting in five of the sets (excluding SYM13 and SYM15) of Figure 16.

The INORABOVE assertions are reduced to eleven, with one assertion for each item except for DOCNAME. Two assertions for DOCNAME remain. They are
INORABOVE(DOCNAME,DOCTOR,–),
INORABOVE(DOCNAME,PATIENT,–).
The designer resolves these by constructing the SYM12 record and the SYM13 and SYM15 sets.

The fact that it was necessary to generate a dummy record may signal the presence of a conflict in the queries (queries one, two and three in this case). On the other hand, if all queries are correct, then it is indeed desired to recognize both a one-to-one and a one-to-many relationship between patients and doctors. This structure will distinguish the principal doctor for a patient from the other doctors on the case.

Note that the designer is able to derive a recursive structure after all! This is a fortuitous accident, because the designer has no explicit knowledge of such structures and cannot design one in a more compact style.

Acknowledgment. The assistance and encouragement of my thesis advisor, Jack R. Buchanan, is gratefully acknowledged.

References

1. CODASYL, Codasyl Data Base Task Group April 71 Report. Available from ACM, New York City; from IFIP Administration Data Processing Group, Amsterdam, and from BCS, London.
2. Gerritsen, R. The relational and network models of data bases: bridging the gap. Proc. 2nd USA-Japan Computer Conf., 1975. AFIPS Press, Montvale N.J.
3. Gerritsen, R. Understanding data structures. Ph.D. Th., Carnegie-Mellon U., Pittsburgh, Pa., 1975.
4. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 6*, (June 1970), 377–387.
5. Lavallee, P.A., Ohayon, S., and Sauvain, R. DMS data base strategies for interrogation and update. Xerox Tech. Rep., undated.
6. Bachman, C.W. Data structure diagrams. Data Base (Quart. newsletter of ACM-SIGBDP) 1, 2 (1969).
7. Buchanan, J.R., and Luckham, D.C. On automating the construction of programs. Stanford AI Project Memo, Stanford U., *Calif., 1974.*
8. Buchanan, J.R. A study in automatic programming, Ph.D. Th., Stanford U., Calif., 1974.
9. Sussman, G. J., and Winograd, T. Micro-Planner reference manual. MIT Project MAC Report, 1972.

Management/ Database Systems
H. Morgan Editor

CONVERT: A High Level Translation Definition Language for Data Conversion

Nan C. Shu, Barron C. Housel, and Vincent Y. Lum
IBM Research Laboratory, San Jose

This paper describes a high level and nonprocedural translation definition language, CONVERT, which provides very powerful and highly flexible data restructuring capabilities. Its design is based on the simple underlying concept of a form which enables the users to visualize the translation processes, and thus makes data translation a much simpler task.

"CONVERT" has been chosen for conveying the purpose of the language and should not be confused with any other language or program bearing the same name.

Key Words and Phrases: data conversion, data restructuring, data translation, database reorganization, translation definition, utility program, programming languages, nonprocedural languages

CR Categories: 3.50, 3.75, 4.29, 4.4, 4.9

I. Introduction

In an overview of the subject [1], the authors have described a general model for data conversion. In this model two essential tools are required to execute the data conversion process: (1) a data definition language to describe the source and target data structures, and (2) a translation definition language to specify the mapping of instances from a set of source files to a different set of target files.

A version of this paper was presented at the ACM SIGMOD Conference on the Management of Data, San Jose, Calif., May 14–16, 1975.

Author's address: IBM Research Laboratory, Monterey and Cottle Roads, San Jose, CA 95193.

In report [2], a language for data description, DEFINE, has been fully specified. This language is capable of describing most linearized data structures; it uses the input/output data format in the conversion model given in [1]. The process of actually obtaining linearized files from source data or creating actual data from them is discussed in the overview paper [1].

The present paper discusses the goals and concepts underlying the design of a translation definition language, CONVERT, and presents the salient features of the language. For a more complete discussion, refer to [3, 4].

II. Design Criteria and Underlying Concepts

With the rapid growth of the computing field, there is a growing need to convert data for various reasons. As pointed out in [5], some of the common reasons are: (1) switching to a different hardware environment; (2) conversion from a conventional file system to a database system; (3) conversion from one database system to another; and (4) a change in application requirements. Some conversions can be fairly simple, others complex. For example, reformatting, a change in the physical and encoding properties of data such as from ASCII to EBCDIC, is simple, while restructuring, which involves changes in the data structures, can be very complex.

In recent years there has been considerable interest in developing a generalized data translation methodology [6–11]. Emphasis so far has been placed on designing a data definition language for describing the logical and physical aspects of data in sufficient detail for a wide variety of data collections. Hopefully, statements in this language can then be used as a driver for data translation. The need for a translation definition language has been recognized; however, efforts in developing such a language have been limited. This is perhaps because most of the attention has been focused on situations where either reformatting is the prime requirement or data restructuring is so simple that each instance of the source is mapped to an instance in the target [11]. The formation of a single target instance from several source files or the derivation of new data from source to form target database according to certain criteria has not been studied.

With the advent of new software systems and expanding applications, data conversion involving extensive and selective restructuring is becoming more and more common. Therefore one of the goals in our design of the translation definition language CONVERT is to provide powerful and flexible restructuring capability in the language.

In designing CONVERT, we have assumed that users of the translation definition language, the translation analysts, are familiar with the logical aspects of their data, know what they want to be done, but do not want to be burdened with the details of how to accomplish it. We also assume that translation analysts have adequate

Fig. 1. (a) A tree graph of hierarchical data.
(b) A Form representation of the same data.

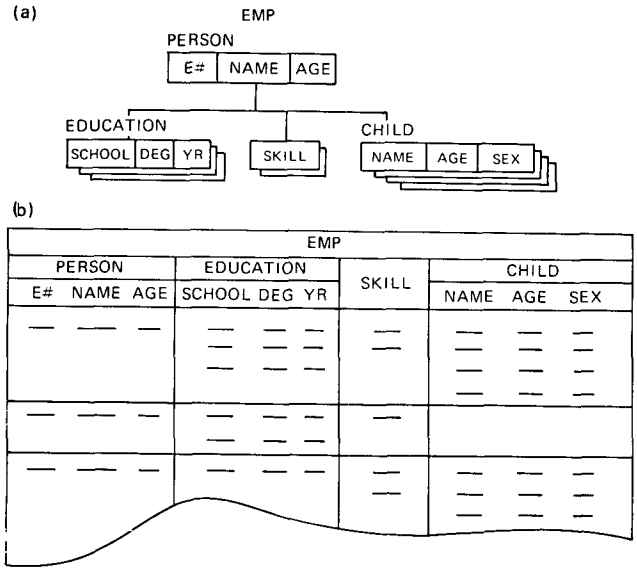


Fig. 2. A PERSONNEL Form.

E#	NAME	EDUCATION			SAL	KIDS	
		SCHOOL	DEG	FIELDS		KNAME	AGE
1	JONES	A B	– B	CS CS	10K	MARY JACK SUE	10 8 5
2	SMITH	A C	B M P	BIO CHEM BIOCHEM	20K	JACK	7
3	DOW	A	B	MATH CS	15K		
4	CARY	D B	B M	CHEM CHEM	18K	MARY	6
5	JONES	C B D	B B P	MATH PHYSICS MATH PHYSICS	25K	JILL SUE JOHN	11 5 3
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•

programming disciplines so that they are willing to follow the syntactic rules of a language. However, they are not mathematically oriented and they do not appreciate semantics in mathematical terms. We have therefore set out to make the language high level, nonprocedural, easy to learn, and simple to use for this type of user.

Our approach is based on a few simple concepts. We started out by exploring the possibility of using the relational model [12, 13] for data translation purposes. However, the formulation of the relational model

requires that relations be at least represented in the first normal form (all simple domains) before any of the relational operators be applied. This constraint requires transformations to be performed on existing data which may not be at all pertinent to the goals of the translation analysts.

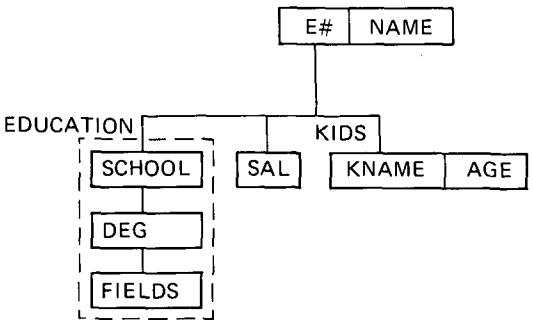
Bracchi et al. [14] in designing a language (COLARD) for a relational database management system, decided to allow the definition of nonnormalized relations. However, because COLARD is based on first-order predicate calculus, one needs mathematical sophistication to use the language, in spite of the English flavor injected into it. Furthermore, while COLARD is specifically conceived to exploit the capabilities offered by the relational model of data, it is not well suited for meeting the requirements of data conversion. For instance, in the "join" operation the generalized n-tuples of two sources are joined which "have the *same* values for the domain." This constraint was found to be too restrictive for conversion purposes.

An in-depth study of the conversion problem convinced us that the translation definition language should be designed to handle all kinds of data structures with equal facility. Furthermore, the language should be developed along some notions familiar to the translation analysts. Since hierarchical data is believed to be the most abundant form of existing data, it plays a dominant role in the development of our translation definition language.

It is envisioned that a translation analyst can view his data in terms of Forms. In Section V, we illustrate the perception of network structure in terms of Forms. But first let us introduce informally the notion of a "Form." (For a more rigorous discussion, refer to [1].) A *Form* is a two-dimensional representation of hierarchical data which reflects the images of data instances. Headings of a Form show the sub-Form (or group) names and field (or item) names by which the various components of data can be referenced. Entries in a Form show the values (or instances) of data items under the appropriate field (or column) headings. For example, one may view a conventional Cobol record with repeating groups or a tree graph [as shown in Figure 1(a)] as a Form [sketched in Figure 1(b)]. Note that both the Cobol declaration and the tree graph show only the data structure while the Form provides a convenient means to visualize the instances.

In practice, it is not necessary to fill in the Form. However, a sketch of a Form [as shown in Figure 1(b)] or a partially filled sample Form showing the headings and some typical occurrences of the data items (Figure 2) will enable the user to visualize the mapping process required in order to compose the target(s) from the source(s). If desired, the user can include additional visual aids in the Form's outlay to increase his perception of the data structure. For example, he may use parentheses to designate a repeating group.

In our approach we have assumed that a user is familiar with his data and therefore knows his data's characteristics. As a matter of fact, he may be the same person who used the data description language DEFINE [2] to describe these characteristics. Nevertheless, it would be helpful to have a precise picture of the hierarchical relationship in order to understand precisely the meanings of the Form operations. For this purpose, we suggest the *hierarchy graph* and some terminologies to describe the hierarchical relationship. In the PERSONNEL file (Figure 2), for example, one may perceive the following hierarchy graph. (Note. An abstraction of the hierarchy graph, that is, a hierarchy graph in machine readable form, is described as TEMPLATE in [2].)



The field names are stated inside the boxes, and the group names outside. The placement of the fields in the graph, from top to bottom and left to right, reflects the organization of the source file. Note that a conventional tree graph does not completely describe the information in this hierarchical structure. We have to use a box of broken lines to indicate the fact that SCHOOL, DEG, and FIELDS can be referred to as a group, namely EDUCATION. Thus a hierarchy graph is simply a tree graph modified to allow the naming of a subtree.

Groups can be formed from fields in the same or different levels. For example, KIDS is a group name for KNAME and AGE; EDUCATION is a group name for SCHOOL, DEG, and FIELDS. However, in the case of KIDS, not only do KNAME and AGE belong to the same group, but they are at the same level. In the case of EDUCATION, on the other hand, SCHOOL, DEG, and FIELDS belong to the same group, but are not at the same level. In terms of hierarchical relationship, SCHOOL is the parent of DEG, and DEG, in turn, is the parent of FIELDS.

We use the term *ancestors* of X (where X could be either an item or a group) to denote the generations of parents along the hierarchical path leading to X. Similarly, we use the term *descendants* of X to denote the generations of children along all hierarchical paths leading from X. Thus ancestors of FIELDS are DEG, SCHOOL, and E#; descendants of E# are EDUCATION (which includes SCHOOL, DEG, FIELDS), SAL and KIDS (which includes KNAME and AGE).

Similarly, we use the term *siblings* to refer to groups and/or items at the same hierarchical level. In the special case where siblings belong to the same group, we call them *twins*. Thus SAL, KIDS, and EDUCATION are siblings while KNAME and AGE are twins. By the same token, NAME is the sibling and not the twin of E# because they cannot be referred together under a group name.

These terms will be used in the discussion of some of the Form operations. It is important to note, however, that here we are talking about the structural *units* in the hierarchy schema. When we discuss Form operations, we are concerned with *instances* of these units.

Another notion which is useful when we discuss operations on instances is a (horizontal) section of a Form. We use the term *section* to denote an instance of the key field of a Form and all information dependent on it (i.e. its twins, siblings, and descendants). In the PERSONNEL Form, for example, where E# is the key field, an instance of E# and its dependents, NAME, EDUCATION, SAL, and KIDS, constitutes a (horizontal) section of the Form. Five sections are shown in Figure 2.

Fig. 3. Examples of Forms.

PTS

P#	DES	S		
		S#	CN	UC
2	X	4	AB	5
		2	BB	4
3	XX	4	AB	2
		1	XB	3
7	Y	7	C	7

INV

P#	QH
2	10
3	17
4	5
7	20

POR

PO	S#	P#	QR
1	4	2	15
1	4	3	2
3	7	7	3

SUP

CN	S#	CA
AB	4	SJ
BB	2	MV
C	7	SF
D	3	LA
XB	1	SJ

III. General Description of the Language

We shall now proceed to describe the language. Actually there are two broad categories of translation definition in CONVERT: data mapping (or restructuring), and data validation. Since the primary purpose of data conversion is to construct target data from various parts of source data, our emphasis is on data mapping. Experience has suggested, however, that recognition of invalid data is a necessary part of the process, since erroneous data is not only unwanted but can also cause much of the difficulty in data conversion. Data validation is discussed in [1–3]. In this paper, we concentrate on mapping specification.

The data mapping and restructuring facilities in CONVERT are provided by a set of Form operators. The Appendix shows a list of the Form operators and their formats as currently defined. They include component extraction, **SELECT, SLICE, GRAFT, CONCAT, MERGE, SORT, ELIM-DUP, CONSOLIDATE,** a set of built-in functions (**SUM, MAX, MIN, AVG,** and **COUNT**), assignment, and **CASE**-assignment. The meanings and uses of some of the more interesting Form operations will be discussed in Section IV. This section depicts only the general characteristics of the various elements in the language.

Each of these Form operators operates on one or more Forms (or their components) and produces a Form as a result. The resultant Form can then be used as operand for another Form operation. Except for assignment and **CASE**-assignment, the formats of which will be discussed in Section IV all Form operations can be nested, and all have the same general format:

Operator (Operands [options] [: Specified conditions])[1]

In describing the operands, we use the following notation:
F denotes a Form which could be either a Form name or the result of a Form operation;
f denotes a field (i.e. a column in a Form);
C denotes a component of a Form, which could be either a field or a sub-Form;
EXPR denotes an arithmetic expression derivable from the fields of a Form. To be more specific, EXPR could be any of the following: (1) a constant, (2) a field name, (3) a built-in function (e.g. **SUM, MAX, MIN, AVG, COUNT**), (4) an expression derivable from 1, 2, or 3 above, or recursively, a derived expression enclosed in parenthesis, using +, −, *, / as arithmetic operators, (5) a sub-Form or group name. (Note that a sub-Form is not allowed to be an operand in arithmetic operation.)

As a rule, the order of appearance of the components or EXPR in the specification determines the component order in the resulting Form.

It may be interesting to note that EXPR, as described above, not only provides the arithmetic capability with which new data items can be computed, it also provides a basis for specifying the conditions for the Form operation to take place. In general, the *specified conditions* (SC) can be expressed as an SC-EXPR, which is defined as logical factors connected by **AND**(s) and/or **OR**(s). A logical factor can be (1) an EXPR compared with another EXPR, or (2) an EXPR com-

[1] The square brackets, [ ], are metasymbols denoting that the enclosed is optional.

pared with **ANY OF** a one-column Form, or (3) an EXPR compared with **ANY OF** a list of single values. The permissible comparison operators include $=$, $\neq$, $>$, $<$, $\neg >$, $\neg <$, $\geqq$, and $\leqq$.

A logical factor is assigned a value of *true* or *false* according to the result obtained from evaluating the comparison. The evaluated logical factors are **AND**ed or **OR**ed together as specified to determine the final *true* or *false* value. Unless parentheses are used to specify the priorities of evaluation, the logical factors are evaluated in left to right order. A Form operation will be executed only if the SC-EXPR yields a *true* result. Note that the conditions specified in an SC-EXPR are not restricted to fields in one Form. Inter-Form conditions may be specified as well.

IV. The Form Operations: Detailed Description

We shall now describe some of the Form operations in detail. For each operation, we shall show the syntax and discuss its function (namely, its meaning and its use). We shall use some simple examples to illustrate these operations. Unless otherwise stated, the examples will be drawn from the source files shown in Figure 3 or the PERSONNEL Form shown in Figure 2.

In Figure 3, PTS is a parts-supplier file, where P#, DES, S#, CN, and UC stand for part number, description, supplier number, company name, and unit cost, respectively. INV is an inventory file containing P# (part number) and QH (quantity on hand). POR is a purchase order file, consisting of PO (purchase order number), S# (the supplier number), P# (the part number), and QR (the quantity requested). SUP is a supplier file having CN (company name), S# (supplier number), and CA (company address).

1. Assignment

Assignment takes the result of the operation(s) specified on the right-hand side of the assign operator ($\leftarrow$) and assigns it to the Form named on the left-hand side. Normally, fields in the target inherit the field-names from the source of the assignment. If one wishes to explicitly name the fields in the target, one may specify them on the left-hand side. For example, the statements I1 $\leftarrow$ POR(P#, S#, QR) and I2(SUPPLIER, PART) $\leftarrow$ POR(S#, P#) produce a Form I1 with column headings P#, S#, QR and a Form I2 with column headings SUPPLIER and PART.

2. SELECT ([EXPR$_1$, . . . EXPR$_n$] FROM F [, ...] [: SC])

This operation selects part(s) of a Form if the specified conditions are satisfied. For example, let us suppose that we want to create a new file consisting of part numbers for the parts supplied by suppliers located in San Jose, together with the corresponding part descriptions and their suppliers' code numbers and names. This could be achieved by stating the **SELECT** operation as follows:

SELECT(P#, DES, S#, CN **FROM** PTS :
PTS.S# = SUP.S# **AND** SUP.CA = 'SJ');

In this case, PTS is the source file from which a target Form consisting of P#, DES, S#, and CN is to be constructed. However, not all instances in the source file will produce an image in the target because we are interested in only those instances where the suppliers are in San Jose. Since the information about the location of a supplier appears only in the SUP file, we must find the tie between the PTS Form and the SUP Form through the use of some common information, which in this case is S#. Hence we have the specified conditions stated as shown. The resulting Form is as follows:

P#	DES	S#	CN
2	X	4	AB
3	XX	4 1	AB XB

In a way, the translation analyst can visualize the selection process as scratching out the items that he does not want. It resembles what he might do with an image of a Form on a scratch pad. Take the Personnel Form in Fig. 2 for example. **SELECT** (**FROM** PERSONNEL: DEG = 'P') could be viewed as follows: The source of our information is the PERSONNEL Form. The translation analyst is interested in all fields in that Form, so he puts the image of PERSONNEL on the scratch pad. He scans down the column of DEG, since DEG is the field which determines whether the source to target mapping should take place. For each instance of DEG encountered, he checks to see if the value is P. If the test fails, he would scratch out that instance of DEG, along with its twins and descendants, if any. (In our example, DEG has no twins, but it has a descendant FIELDS.) He repeats this for all occurrences of DEG.

He then proceeds to examine the parent instance for each occurrence of DEG. For a particular parent instance, if no instance of DEG survived on the scratch pad, he would scratch out that parent instance, along with its twins, if any, and proceed to examine the next ancestor. He does this until all the ancestors have been examined. If an instance at the top level is eliminated, all information dependent on it is scratched. Thus the result of **SELECT**(**FROM** PERSONNEL: DEG = 'P'); is shown in the following Form:

		EDUCATION				KIDS	
E#	NAME	SCHOOL	DEG	FIELDS	SAL	KNAME	AGE
2	SMITH	C	P	BIOCHEM	20K	JACK	7
5	JONES	D	P	MATH PHYSICS	25K	JILL SUE JOHN	11 5 3

Because the **SELECT** operation not only exemplifies the concepts underlying all of the Form operations, but also gives some insight on the effect of ancestors, descendents, siblings, and twins, we have gone into detail to illustrate how items are conditionally selected. Conditional selection, however, is not the sole function provided by the **SELECT** operator. In addition, it also provides a facility to derive new data. As mentioned in Section III, computations can be performed on selected fields. Again, take the PERSONNEL Form (Fig. 2) as an example. The SAL field represents the monthly earning of each employee. Suppose we wish to create a target file TF, consisting of E#, NAME, and each person's weekly salary. This can be accomplished in the following statement:

TF(E#, NAME, WAGE) ← **SELECT**
(E#, NAME, SAL * 12/52 **FROM** PERSONNEL);

3. SLICE($f_1, \ldots, f_j$ **FROM** F)

The **SLICE** operation provides the capability to produce one row for each instance of f_j. To be more specific, the **SLICE** operation produces, for each occurrence of f_j, a row consisting of the corresponding instances of $f_1, \ldots, f_j$ where f_i, for $1 < i < j - 1$, must be an ancestor of f_j, a twin of the ancestor, or a twin of f_j. For example, the result of **SLICE**(E#, DEG, FIELDS **FROM** PERSONNEL) is shown in Figure 4.

Thus the **SLICE** operation provides a convenient means to produce relational tables from hierarchical structures. It should be noted, however, that since $f_1, \ldots, f_j$ are fields along *one* hierarchical path, each **SLICE** operation produces only one relational table. It is often the case that more than one relation is encompassed in a hierarchical structure; accordingly, more than one **SLICE** operation would be required to transform completely a hierarchical structure into a set of relational tables.

4. SORT(F[**BY** [ASCENDING / DESCENDING] $f_1, f_2, \ldots, f_n$][**WITHIN PARENT**])

The SORT operation sorts the instances of a Form in either ascending or descending order of $f_1, f_2, \ldots, f_n$ where $f_1, \ldots, f_n$ are members in the same path of a tree. The sort fields $f_1, f_2, \ldots, f_n$, when specified, should be listed from left to right in order of decreasing significance, regardless of whether they are ascending or descending. If not specified, the sort order will be assumed to be ascending and all fields of the Form, in left to right order, will be considered as sort fields of decreasing significance.

If the **WITHIN PARENT** clause is specified, sorting will be performed over the instances of the sort fields without effecting the sequences of the parent instances. Take F4, for example: F4A ← **SORT**(F4 **BY** P# **WITHIN PARENT**) produces F4A (Figure 5).

On the other hand, if **WITHIN PARENT** is not specified, sorting will be performed over the entire file.

Fig. 4.

E#	DEG	FIELDS
1	–	CS
1	B	CS
2	B	BIO
2	M	CHEM
2	P	BIOCHEM
•		
•		
•		

Fig. 5.

F4

S3	P#
A	10
	11
	12
C	15
	14
D	10
	12
B	13
	10

F4A

S#	P#
A	10
	11
	12
C	14
	15
D	10
	12
B	10
	13

Fig. 6.

F7

S#	P#	QTY
S1	P1	3
S1	P2	2
S1	P3	4
S1	P4	2
S1	P5	1
S1	P6	1
S2	P1	3
S2	P2	4

F7A

S#	P#	QTY
S1	P1	3
	P2	2
	P3	4
	P4	2
	P5	1
	P6	1
S2	P1	3
	P2	4

In this case, the most significant sort field *must* be either the key field of the Form (on which all the other fields are dependent) or a field which has a 1:1 correspondence with the key field. The units for sequencing will be the horizontal sections of the Form.

5. CONSOLIDATE (F **FOR UNIQUE** $\{^{f_1, f_2, \cdots}_{(f_1, f_2, \cdots), (f_a, f_b, \cdots)\cdots}\}$)

A Form is a representation of hierarchical data. In its pure form, it reflects the hierarchical relationship among data instances. However, Forms are not always pure. In F7 (Figure 6), for example, there are repeated occurrences of S1, S2, S3, and S4 in the field S# which are superfluous in expressing a hierarchical relationship. There could be many reasons for them to appear in the

Fig. 7.

F8

X	Y	Z	W
A	P1	10	WA1
	P2	20	WA2
	P3	30	
B	P1	15	WB1
	P4	25	
	P5	35	
A	P1	2	WA3
	P3	4	WA1
	P5	6	
B	P4	30	WB1
C	P5	20	WC1

F8A

X	Y	Z	W
A	P1	10	WA1
		2	WA2
	P2	20	WA3
	P3	30	
		4	
	P5	6	
B	P1	15	WB1
	P4	25	
		30	
	P5	35	
C	P5	20	WC1

Fig. 8.

P#	DES	S			QH
		S#	CN	UC	
2	X	4	AB	5	10
		2	BB	4	
3	XX	4	AB	2	17
		1	XB	3	
7	Y	7	C	7	20

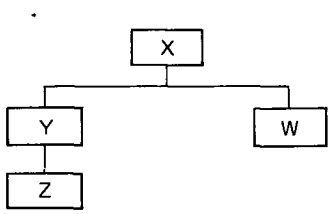
Form: Maybe they are inherited from the source. Maybe they are designed for the target, or maybe they are the result of a **SLICE** operation. F8 (Figure 7) is another example. There are duplicate instances of A and B in field X; and for a unique value of X (e.g. A or B), there are duplicate occurrences of Y and W. Again it is not a pure Form in the sense that the appearances of the instances in the Form do not reflect the hierarchical relationship in a clear way.

Just as there are reasons why redundancies appear, there may be good reasons to have them removed. In other words, we might wish to transform F7 and F8 into F7A and F8A respectively. To accomplish this, the **CONSOLIDATE** operation may be used. It has the following general format:

CONSOLIDATE(F **FOR UNIQUE**(f_1, f_2, . . .), (f_a, f_b, . . .), . . .)

Note that as in the case of **SORT**, the set of fields within each pair of parentheses (e.g. f_1, f_2, . . .) must be fields in the same hierarchical path. The specification of **FOR UNIQUE**(f_1, f_2, . . .), (f_a, f_b, . . .), . . . requeses consolidation to be performed along two or more hierarchical paths, one for each of the parenthesized list; but each of the paths must start from the key field (i.e. $f_1 = f_a$). When the **CONSOLIDATE** operation is required on only one hierarchical path, a simpler format, **CONSOLIDATE**(F **FOR UNIQUE** f_1, f_2, . . .), may be used.

To be more specific, the **CONSOLIDATE** operation transforms a Form F by grouping the data according to unique values of f_1, then for a unique value of f_1, groups its descendants by the unique values of f_2, and so on. Thus the statement for transforming F7 into F7A is **CONSOLIDATE**(F7 **FOR UNIQUE** S#), and the statement for transforming F8 into F8A is **CONSOLIDATE**(F8 **FOR UNIQUE**(X, Y), (X, W)). The hierarchy graph for F8A is shown below:



6. **GRAFT**(F_1, F_2, . . . **ONTO** F_n[**AT** f] [: SC])

GRAFT provides a means to combine two or more Forms into one Form when specified conditions are satisfied. It produces Cartesian Products when : SC is omitted.

In general, the conditions to be satisfied can be stated as an SC-EXPR as described in Section III. Since **GRAFT** operates on two or more Forms, it should be apparent that the SC-EXPR if specified should include at least the logical factors which serve to tie the Forms together. For example, suppose we wish to form one file from the PTS and INV files such that the resulting file will have the information of the PTS file plus the quantity on hand (QH) obtained from INV. This can be stated as follows:

GRAFT(INV **ONTO** PTS: PTS.P# = INV.P#);

Here the SC-EXPR serves as a tie between the PTS and the INV files. There are two tying fields: P# of PTS and P# of INV. Only the one in the Form after **ONTO** will appear in the resulting Form. The result is shown in Figure 8.

This way of stating conditions to be satisfied is useful in most of the cases. However, there are situations where some of the data exists only in some (not all) of the files that we are interested in. In the PTS and INV files, for example, P# = 4 exists in INV but not in PTS. By stating PTS.P# = INV.P# as the satisfying condition, we have excluded P# = 4 from our new file. What if for some reason we wish to include all P#'s in our new file, leaving the missing information blank? To achieve this, we use the **PREVAIL** clause to specify the conditions.

The **PREVAIL** clause, in general, takes the following format:

f_1, f_2, . . . **PREVAIL**[f_j, f_k, . . . , f_n]

where f_1, . . . , f_n are the names of the fields whose values are to be "matched." To avoid the need for synonym specification, one may use in the **PREVAIL**

clause a qualified field name for each of the Forms participating in the **GRAFT** operation. The names on the left-hand side of the key word **PREVAIL** are considered to be the *prevailing* fields. The union of the instances of the prevailing fields determines the instances included in the resulting Form. In the PTS, INV, and POR files for example, the statement **GRAFT**(INV, POR(P#, QR) **ONTO** PTS: INV.P# **PREVAIL** PTS.P#, POR.P#); produces the result shown in Figure 9.

In the cases where more than one level of keys are required for matching, the **PREVAIL** clauses may be connected by **AND** in the following manner.

$f_1, f_2, \ldots$ **PREVAIL** $f_f, f_j, \ldots$
AND $f_a, f_b, \ldots$ **PREVAIL** $f_m, f_n, \ldots$

where each **PREVAIL** clause specifies one level of fields to be matched.

7. Built-in-Functions:

$$\left\{\begin{matrix}\textbf{SUM}\\ \textbf{MAX}\\ \textbf{MIN}\\ \textbf{AVG}\\ \textbf{COUNT}\end{matrix}\right\} (\text{f } \textbf{IN F}[\textbf{FOR UNIQUE } f_1, \ldots f_n] [: \text{SC}])$$

The built-in functions compute the sum, maximum, minimum, average, or count of the instances of a certain field f in a Form F where the specified conditions are satisfied. They all have exactly the same format and operate in exactly the same manner. If the **FOR UNIQUE** $f_1, \ldots, f_n$ option is taken, the computation will be performed over instances of f for unique values of $f_1, \ldots, f_n$ where $f_1, \ldots, f_n$ must be ancestors of f. If there is no **FOR UNIQUE** clause stated, the computation will be performed over all instances of f in the Form.

The following examples illustrate the application of built-in functions.

Examples. Given F as in Figure 10. Then
the **SUM** (C **IN** F) result is 2 + 15 + 7 + 9 + 22 + 14 + 17 + 20 + 32 + 24,
the **COUNT** (C **IN** F : C < D) result is 7, and
the **SELECT**(A, **SUM** (C **IN F FOR UNIQUE** A) **FROM** F) result is 2 + 15 + 7 for Q, 9 + 22 for R, etc.

8. CASE Assignment

Every one of the Form operators discussed so far performs one uniform operation over all instances of the relevant Form(s). **CASE** Assignment, on the other hand, allows varied operations to be performed over different instances. These varied operations must produce homogeneous results to be assigned to the resulting Form. The variation is dependent on some prescribed tests either on the value of a single instance of a field or on a set of values of a specific field for unique parent or ancestors. Hence there are two formats for the **CASE** assignment. We shall discuss them in turn.

Fig. 9.

		S				
P#	DES	S#	CN	UC	QH	QR
2	X	4	AB	5	10	15
		2	BB	4		
3	XX	4	AB	2	17	2
		1	XB	3		
4	—	–	—	–	5	—
7	Y	7	C	7	20	3

Fig. 10.

		F		
A	B	C	D	E
Q	1	2	3	4
		15	6	
		7	8	
R	2	9	10	11
		22	13	
S	3	14	15	16
T	1	17	18	19
		20	21	
		32	23	
	2	24	25	26

Format 1. The first format is

F ← **CASE**(f COP $v_1, v_2, \ldots, v_n$[, **OTHERS**)]
($F_1, F_2, \ldots, F_n$[, F_{n+1}]);

As usual, F and f denote a Form and a field respectively; COP denotes a comparison operator. v_i denotes a single value, defined as follows:

⟨Single-Value⟩ ::= ⟨Value⟩
| ⟨Single-Value⟩ **OR** ⟨Value⟩
⟨Value⟩ ::= ⟨Literal⟩
| **ANY OF** ⟨FORM⟩

and ⟨FORM⟩, in turn, is either a Form name or a nestable Form operation representing a one-column Form.

With this format, Assignment is allowed to be varied according to the value of an occurrence of the specified field f. For each instance of f, its value is compared with v_i (where $1 \leq i \leq n$) in the left to right order until a *true* result is obtained from the evaluation (e.g. $f \leqq 70, 80, 90$ is evaluated as $f \leqq 70$, $70 < f \leqq 80$, $80 < f \leqq 90$, in that order). As soon as the result of evaluating an instance of f against v_i is *true* (i.e. the **CASE** test is satisfied), the corresponding F_i will be activated to provide the source for assignment.

F_i could be any of the Form operations (except the assignment operation) that we have defined. They have exactly the same functions as we described earlier. The scope of these operations, however, is limited to those instances satisfying the **CASE** tests. For this reason, we use the italic *F* (instead of F) to denote the Form operations effective for **CASE** assignment.

Fig. 11.

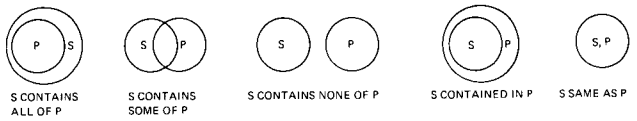


Fig. 12.

SUPPLIER

S#	P#
A	10
	11
	12
B	10
	12
	13
C	14
	15
D	10
	11
	12
	13

PARTS

P#	DESC
10	X
11	Y
12	Z

Fig. 13.

Source 1:

DEPT

D#	DNAME	DMGR	EMP		BUDGET	
			E#	ENAME	DOLLARS	YR

Source 2:

EMP

E#	ENAME	SEX	EDUCATION		
			DEG	YR	SCHOOL

Target T

TDEPT

D#	DNAME	DMGR	CURRENT_BUDGET

SUPPORT			SR			SC		
E#	ENAME	SEX	E#	ENAME	SEX	E#	ENAME	SEX

(NO DEGREE) (ANY DEGREE>15 YRS) (ALL DEGREE≤15 YRS)

Furthermore, v_i and F_i must be paired. If the optional pair of [, OTHERS] and $[,F_{n+1}]$ is not specified, no operation is performed when all tests specified in f cop $v_1, \ldots, v_n$ fail.

Take the following example. Suppose we have the source Form SF and we wish to change the entries of FEMALE and MALE in the SEX field into SEXCODE where 0 represents female and 1 represents Male. This can be achieved with the following statement:

```
T(E#, SEXCODE)
        ← CASE (SF.SEX = 'MALE', 'FEMALE')
          (SELECT (E#, '1' FROM SF),
           SELECT (E#, '0' FROM SF));
```

Format 2. The second format is

$F \leftarrow$ **CASE** ((**FOR UNIQUE** $f_1, f_2, \ldots$)
f $Mre_1, Mre_2, \ldots, Mre_n$ [,**OTHERS**])
$(F_1, F_2, \ldots, F_n [,F_{n+1}])$.

While the first format allows an assignment to be varied according to the single value of an occurrence of a specific field, the second format allows the assignment to be varied according to a set of values of a specific field f for an occurrence of unique parent or ancestors. To illustrate this point, let us use S to denote a set of values of field f for unique $f_1, \ldots, f_n$, and use P to denote another set of values comparable to the menbers of S. When we compare the members of these two sets, any one of the situations shown in Figure 11 may occur.

In other words, when we describe the members of S as compared to P, there are five *Member relationships* (Mr) that we can draw upon: **CONTAINS ALL OF** P, **CONTAINS SOME OF** P, **CONTAINS NONE OF** P, **CONTAINED IN** P, or **SAME AS** P. These Member relationships may be connected by the key word **OR** to form a *Member relationship expression* (Mre). For example, **CONTAINS SOME OF** P **OR SAME AS** P is an Mre composed of two member relationships: **CONTAINS SOME OF** P and **SAME AS** P.

The second format of the **CASE** assignment allows a series of these member relationship expressions to be specified. For unique $f_1, f_2, \ldots$, the set of members of f will be tested against these expressions in the left to right order. As soon as a *true* result is found when evaluating f against Mre_i, the corresponding F_i will be activated to perform the operation. We refer to the expression f $Mre_1, Mre_2, \ldots, Mre_n$ as *Member tests.*

For example, suppose we have two Forms, SUPPLIER and PARTS, as shown in Figure 12.

To find the suppliers who furnish exactly the same parts as listed in the PARTS file, one may specify:

```
T(S#) ← CASE((FOR UNIQUE SUPPLIER.S#)
            SUPPLIER.P#
            SAME AS PARTS(P#))
            (SELECT(S# FROM SUPPLIER));
```

With this specification, there is only one Mre in the Member test. For each unique S# in SUPPLIER, the set of P# is compared with the set of P# in PARTS. In the case of S# = A, the set of SUPPLIER.P# is (10, 11, 12), which is exactly the same as the set of P# in PARTS. Thus the corresponding F_i is activated, i.e. S# is selected from SUPPLIER and the result is assigned to the target. In all other cases, the result of testing is *false* and no action is taken.

We have described in detail the syntax and the semantics of the Form operations. We shall now use an example to illustrate how these Form operators can be used together to specify the mapping from source to target. The source and target structures are shown in Figure 13. Translation specification may be stated as

follows:

1. IEMP(D#, E#, ENAME, SEX, DYR)
   ← CASE(EMP.EDUCATION.DEG = NULL, OTHERS)
   GRAFT(SELECT(E#, ENAME, SEX, 0 FROM EMP)
   ONTO DEPT(D#, E#) : DEPT.E# = EMP.E#),
   GRAFT(SELECT (E#, ENAME, SEX, MIN(EMP.
   EDUCATION.YR FOR UNIQUE EMP.E#)
   FROM EMP) ONTO DEPT(D#, E#) : DEPT.E#
   = EMP.E#));
2. SUP ← SELECT(D#, E#, ENAME, SEX FROM IEMP :
   IEMP.DYR = 0);
3. TSR ← SELECT(D#, E#, ENAME, SEX FROM IEMP :
   IEMP.DYR ≠ 0 AND 1974 − IEMP. DYR
   > 15);
4. TSC ← SELECT(D#, E#, ENAME, SEX FROM IEMP :
   1974 − IEMP.DYR ≦ 15);
5. TDEPT(D#, DNAME, DMGR, CURRENT_BUDGET)
   ← SELECT(D#, DNAME, DMGR, DOLLARS FROM
   DEPT : DEPT.BUDGET.YR = 1974);
6. T ← GRAFT(SUP, TSR, TSC ONTO TDEPT : TDEPT.D#
   PREVAIL SUP.D#, TSR.D#, TSC.D#);

In this example of translation specification, statement 1 creates a Form, IEMP, which consists of D#, E, ENAME, SEX, and DYR (i.e. the year an employee obtained his earliest degree, if any) for all employees. Statements 2, 3, and 4 split IEMP into 3 Forms: SUP consists of employees who have no degree, TSR consists of employees whose earliest degrees were obtained more than 15 years ago, and TSC consists of employees who have obtained their degrees within the last 15 years. Statement 5 constructs a Form TDEPT, for all departments, containing D#, DNAME, DMGR, and the current budget (i.e. the amount of DOLLARS for the year 1974). Finally, statement 6 grafts the three groups of employees in SUP, TSR, and TSC onto TDEPT, thus producing the final target Form as desired.

V. Expressing Different Data Structures in Terms of Forms

In Section II we alluded to the fact that other data structures can be expressed via the Forms. We shall now illustrate how this can be achieved using the network shown in Figure 14 as an example.

Each node in the network can be viewed as a Form. For instance, the nodes DEPT and EMPLOYEE are shown as forms in Figure 15.

Each named edge represents a means of connecting two forms. Conceptually, there are two ways to provide these connections. One way is to have the connecting information embedded in one or both of the Forms. Another is to build a Form to represent the information expressed by the edges. In our example, the edges DE and DP are embedded in the DEPT Form, while A and W appear as separate Forms, as shown in Figure 16.

It is important to note that how to express the edge that serves as connection between any two Forms is the user's decision. Conceptually, however, it is possible to adopt the notion of a Form as we described in this paper as a basis for more complex data structures. Thus, although CONVERT is primarily designed for hierarchically structured data, it is applicable to other kinds of data structures as well.

Fig. 14.

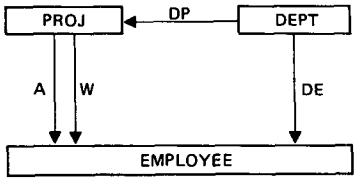


Fig. 15.

DEPT

D#	MGR	P#	E#
55	SMITH	P1	551
		P2	552
		P3	553
			554
			555
			556
54	JONES	P1	541
		P4	542
			543
			544

EMPLOYEE

EMP#	EDUCATION		SKILL
	DEG	YR	
–	–	–	–
	–	–	
–	–	–	–

Fig. 16.

PROJ

P#	LEADER	BUDGET
P1	541	100K
P2	554	200K
P3		50K
P4	542	300K

A

P#	EMP#
P1	541
P2	551
P4	541

W

P#	EMP#
P1	551
	552
	542
P2	553
	554
	555
P3	555
	556
P4	542
	543
	544

VI. Conclusion

In this paper we have described in detail a language for specifying the mapping of the instances of source items, which may be components of one or more files, into instances of target data, which may constitute multiple files.

The language is designed for the class of users who are familiar with the logical aspects of their data, know what they want to be done, but do not want to be concerned with the details of how to accomplish it. It is high level and nonprocedural according to current

standards. In a preliminary study we applied the language to three real-life examples where conversion had been carried out using PL/I. We described the same conversion process using CONVERT without nesting. We then counted the number of data mapping statements in both cases and found that CONVERT required five to twenty times fewer statements than PL/I.

The language provides very powerful [4] and highly flexible restructuring capability. In fact, it has the capabilities required for a general hierarchical database language [15–17]. Although primarily designed for hierarchically structured data, it is applicable to other kinds of data structures as well. It is believed that the language can handle all common processes required in a data translation. Furthermore, the simple underlying concepts enable the users to visualize the translation processes, thus making data conversion a much simpler task.

Appendix. List of Form Operators

1. Component extraction $F(C_1, C_2, \ldots C_n)$
2. Assignment (see p. 561)
3. **SELECT**([EXPR, ... EXPR] **FROM** F [, ...] [: SC])
4. **SLICE**($f_i \ldots f_j$ **FROM** F)
5. **GRAFT**($F_1, F_2, \ldots$ **ONTO** F_n [**AT** f] [: SC])
6. **CONCAT**($F_1, F_2, \ldots$ **ONTO** F_n **AT** f)
7. **MERGE** ($F_1, F_2, \ldots F_n$)
8. **SORT**(F[**BY** [$\substack{\text{ASCENDING}\\\text{DESCENDING}}$] $f_1, f_2, \ldots$] [**WITHIN PARENT**])
9. **ELIMDUP**(F)
10. **CONSOLIDATE**(F **FOR UNIQUE** $\{^{f_1, f_2, \ldots}_{(f_1, f_2, \ldots), (f_a, f_b, \ldots)}\}$)
11. Built-in Functions $\left\{\begin{matrix}\text{SUM}\\\text{MAX}\\\text{MIN}\\\text{AVG}\\\text{COUNT}\end{matrix}\right\}$ (f **IN** F[**FOR UNIQUE** $f_1, f_2, \ldots$] [: SC]
12. **CASE** Assignment (see p. 564)

Notes
[ ] denotes enclosed are optional.
{ } denotes one of the enclosed must exist.
Words in boldface type are reserved words.
WHERE may be substituted for ":".

Acknowledgments. The authors are grateful to W.F. King III for his guidance and encouragement, to D.P. Smith for many helpful discussions and suggestions, and to W.G. Tuel, M.C. Smyly, and G.C. Giannotti for providing real-life examples.

References

1. Lum, V.Y., Shu, N.C., and Housel, B.C. Data translation, Part I. A general methodology for data conversion and restructuring. IBM Res. Rep. RJ 1525, July 1975.
2. Housel, B.C. Lum, V.Y., and Shu, N.C. Data translation, Part II. DEFINE: a non-procedural language for DEFining INformation Easily. Proc. ACM Pacific 75 Conference, Apr., 1975, pp. 62–70.
3. Shu, N.C., Housel, B.C., and Lum, V.Y. Data translation, Part III. CONVERT: a high level translation definition language for data conversion. IBM Res. Rep. RJ 1515, Feb. 1975.
4. Smith, D.P. Data translation. Part IV. Relational Completeness of the translation definition language, CONVERT. IBM Res. Rept. RJ 1527, in preparation.
5. Housel, B.C., Lum, V.Y., and Shu, N.C. Architecture to an interactive migration system (AIMS) Proc. ACM SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., 1974, pp. 157–170.
6. Fry, J.P., Smith, D.P., and Taylor, R.W. An approach to stored data definition and translation. Proc. ACM SIGFIDET Workshop on Data Description and Access, Denver, Colo., 1972, pp. 13–55.
7. Fry, J.P., Frank, R.L., and Hershey III, E.A. A developmental model for data translation. Proc. ACM SIGFIDET Workshop on Data Description and Access, Denver, Colo., 1972, pp. 77–105.
8. Smith, D.P. A method for data translation using the stored data definition and translation task group languages. Proc. ACM SIGFIDET Workshop on Data Description and Access, Denver, Colo., 1972, pp. 107–124.
9. Sibley, E.H., and Taylor, R.W. A data definition and mapping language. *Comm. ACM 16*, 12 (Dec. 1973), 750–759.
10. Merten, A.G. and Fry, J.P. A data description language approach to file translation. ACM Proc. SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., 1974, pp. 191–205.
11. Fry, J.P. Stored data definition and translation approach to the data portability problem. Data Translation Project Rep., U. of Michigan, Ann Arbor, Mich., Feb. 1974.
12. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
13. Codd, E.F. Normalized data base structure: a brief tutorial. Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., 1971, pp. 1–17.
14. Bracchi, G., Fedeli, A., and Paolini, P. A language for a relational data base management. Proc. Sixth Ann. Princeton Conf. on Information Science and Systems, Mar. 1972, pp. 84–92.
15. Senko, M.E., Altman, E.B., Astrahan, M.M., and Fehder, P.L. Data structures and accessing in data base systems. *IBM Systems J 12*, 1 (1973).
16. Altman, E.B., A hierarchic representation independent language (HRIL). I: hierarchy qualification functions. IBM Res. Rep. RJ 1215, May 1973.
17. Fehder, P.L., HQL: a set-oriented transaction language for hierarchically-structured data bases, Proc. ACM Annual Conference, San Diego, California, 1974, pp. 465–472.