y5931482 1323261 1894

VALIDATION OF QUERIES TO A RELATIONAL DATA BASE

Douglas Ray Hainline

Submitted to the Council for National Academic Awards in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Sponsoring Institution:

Thames Polytechnic

May 1986

Theses

005.756 HAI

Validation of Queries to a Relational Data Base

Douglas Ray Hainline

This thesis addresses the problem of preventing users of a data base system from interrogating it with query language expressions which are syntactically and semantically valid but which do not match the user's intentions. A method of assisting users of a relational data base to formulate query language expressions which are valid representations of the abstract query which the user wishes to put is developed.

The central focus of the thesis is a method of communicating the critical aspects of the semantics of the relation which would be generated in response to a user's proposed operations on the data base. Certain classes of user error which can arise when using a relational algebra query system are identified, and a method of demonstrating their invalidity is demonstrated. This is achieved by representing via a graph the consequences of operations on relations. Also developed are techniques allowing the generation of pseudo-natural language text describing the relations which would be created as the result of the user's proposed query language operations.

A method of allowing the creators of data base relations to incorporate informative semantic data about their relations is developed. A method of permitting this data to be modified by query language operations is specified. Pragmatic linguistic considerations which arise when this data is used to generate pseudo-natural language statements are addressed, and examples of the system's use are given.

ACKNOWLEDGEMENTS

The author wishes to acknowledge with gratitude the early helpful advice of Dr Barry Alldred, representing this work's initial external sponsor, IBM. He also is indebted for the long support and encouragement of both of his supervisors, Tom Crowe and Dr Roger Johnson. The latter was a particularly fruitful source of critical and constructive ideas, as well as displaying patience and a faith in the work which at times seemed to surpass all understanding. He finally wishes to thank his wife, Dr Judith Shapiro, for serving both as proof-reader and as a conscript test-subject.

TABLE OF CONTENTS

Section Page

Section Page

Section Page

Section Page

Section Page

SUMMARY

This thesis describes an approach to providing a significant measure of semantic validation for user queries to a relational database. The problem addressed is the question of how to provide assurance to the casual user of a formal query language that a particular query expression constructed by him corresponds to his intentions. The query language chosen to illustrate the approach is relational algebra.

The method expounded in the thesis has been realised in a computer program. This program accepts relational algebra queries typed in by the user, and returns a pseudo-natural language "reverse translation" of the query. The following is an example of the system in operation:

Consider a database consisting of two two-column relations: PSL, associating Persons and the Languages spoken by them; and PRL, associating Persons and the Languages they read. The system described in this thesis can take as input the superficially-similar relational algebra query expressions (PSL % [Person]) - PRL % [Person]) and (PSL - PRL) % [Person] (a difference between two projected relations, and a projection of two differenced relations, respectively) and generate text which will allow the user to see the difference between the semantics of the two queries. The first expression yields persons who speak a language but read none. The second expression returns the broader set (which includes the first group) of persons who speak a language which they do not read.

The methods developed in this thesis to accomplish the "reverse translation" are a novel approach to query validation and constitute an original contribution to knowledge. The layout of the thesis is as follows:

Chapter I places the problem of query validation within the context of validation in general, reviews research into the types of errors that database users make in placing queries, distinguishes among three levels of reference for the term 'query', analyzes queries in terms of their component features, and surveys related work.

Chapter II expounds the foundations of the approach developed in the remainder of the thesis by setting out precise definitions for the relational approach to data base design, and discussing the relevance of the Entity/Relationship approach to the present work. It then summarises the method used (translation of Relational Algebra expressions into information-bearing graphs), and discusses the kind of validating information conveyed to the user by the process.

Chapter III describes the abstract data structure (a directed graph) which is the underlying mechanism through which "reverse translation" is achieved.

Chapter IV continues the exposition of the techniques of reverse translation by describing the effect of each Relational Algebra operator on the graphs attached to the relations upon which these operators are applied. Example queries using each operator demonstrate the pseudo-natural language text generated from the graphs of the relations derived from Relational Algebra queries.

Chapter V takes up the methods needed to deal with "qualified entities" (those which have been affected directly or indirectly by a Relational selection operation).

Chapter VI deals with the methods needed to handle Relational set operations.

Chapter VII provides a set of example queries illustrating various possible user "traps", and demonstrates the application of the techniques developed in previous chapters to each of them.

Chapter VIII concludes with a discussion of the promise and limitations of the approach developed in the thesis, with particular reference to possibilities for further research.

Summary

A list of references is followed by appendices containing listings of the Pascal code used to implement the approach of the thesis, and examples of sessions creating domains and base relations.

CHAPTER ONE

INTRODUCTION AND BACKGROUND

1.1 Introduction

The evolution of computer technology has steadily expanded both the scope of applications for which computers can be used, and the number of people brought into direct contact with computer-based systems. The development of direct-access mass storage technology allowed the modelling of complex systems in data bases, while visual display units allowed new layers of users casual access to this data, mediated by query languages.

The relationships and properties of the objects represented in a database can be modelled via several formalisms. One of the simplest of these, sustaining a large proportion of database-oriented research, is the relational approach, which provides the data model assumed in this thesis.

Each relation in a relational data base is normally intended to represent collections of real-world objects. Each relation also has an "intention" which can be expressed by a user who understands the application and the data base model of it. A user who understands both the semantics of a query language and the intention of the relations which are used as arguments to query language expressions will be able to describe the intention of the relation which is yielded by the query process.

The fact that human beings can "generate" descriptions of derived relations suggests that it would be useful to investigate the possibility that a machine can do the same thing. A machine-implementable method for generating descriptions of derived relations would be a powerful method of validating that the relation derived as the result of a complex query expression was indeed the relation desired by the user.

In pursuit of this goal, this thesis develops a method of constructing, for the base relations of a relational data base, graph structures which can be used to assist users of the data base in formulating query language expressions which are valid representations of the abstract query which the user wishes to make. The base relation graphs can be manipulated by expressions of the relational algebra to produce modified graph structures corresponding to the derived relations which would be produced by the relational algebra expression. These structures can in turn be used to generate pseudo-natural language text for understanding the meaning of the derived relations. This technique gives an end user the possibility of checking that his proposed query language expressions correspond to his intensions.

1.2 Background to Query Language Research

1.2.1 Motivation for research into validation

One of the central trends which has characterised the evolution of computing systems is the expansion of the class of users who can interact directly with the system. Strong economic imperatives motivate this trend. The CODASYL End User Facilties Committee summarises this development in the following words:

> Data processing hardware trends are well known. There continues to be steady and dramatic decreases in costs coupled with equally steady and dramatic improvement in raw performances. The non-hardware costs of data processing, however, are primarily people related costs which are rising due to both inflation and scarcity of trained Programming professionals.... To adapt to these trends, it is desirable to bring the end user into the act in partnership with data processing professionals to shorten the application development cycle. At the same time, it is necessary to develop facilities that enable the end users to do a great deal of their data processing, independent of data processing personnel. [CODASYL EUF 1979]

The Committee distinguished "end users" who "are generally not trained in programming or other data processing technologies" from "system users," and divides the former group into three broad classes.

(1) Indirect users, who deal with the system through other people.

(2) Intermediate end users, who specify the information requirements to be provided by data processing.

(3) Direct end users, who are "functional professionals and their personnel who directly interact with the computer in accomplishing their work. They include accountants, engineers, salesmen,

etc." [CODASYL, _ibid_.]

It is "direct end users" in the CODASYL EUFC sense to whom the research results in this thesis are expected to be relevant.

A large proportion of direct end user/computer system interactions will involve integrated data base management systems, accessed in the _ad hoc_ query mode (excluding modifications of the data base such as updates). In order to perform successful retrievals from a data base, a user must be familiar with the application being modelled, with the data model used to provide a logical organisation of the data, and with the query system used to manipulate the data model.

Familiarity with the application on the part of the user will be assumed in this study, although "familiarity" will not be taken to mean "perfect knowledge". Methods of teaching new users about the semantics of a particular data base model of a real world system are worthy of separate investigation. Some of the techniques developed here could be used to impart information about base relations and domains, but a detailed investigation of this possibility is beyond the scope of this thesis.

The core of the query system used to manipulate the data model will be a relational algebra, similar to the Information System Base Language (ISBL) developed at the

IBM UK Research Laboratories to interface with the Peterlee Relational Test Vehicle (PRTV), a working data base management system. [Todd, 1976]

1.2.2 Validation

1.2.2.1 Overview

Data is a representation via symbols of some aspect of the world. In the typical data processing system data is transformed by programs. The input data and the transforming programs are valid when they represent the world according to our intentions. Human intentions are the ultimate measuring stick for every dimension of validity.

Validity is not necessarily an absolute concept. Data which is valid for one application may be invalid for another due to lack of precision. [Crowe and Jones, 1975] An error in occurences of one kind of data may of much greater consequence than an error in occurences of another kind. (For example, customer bank balance in comparison to customer date of birth). A program may work for normal sets of inputs, but not for unusual sets. A statement about the validity of a program must refer to the range of data for which the program is claimed to be valid. The choice of this range is a human decision.

1.2.2.2 Data Validation

Data is validated by applying logically redundant techniques: it is recorded more than once, perhaps in different modes, and corresponding values matched. It is checked to see if it contravenes known constraints. [Hammer and McLeod, 1975; Hammer, 1976] Where data is processed repeatedly and output to a user community familiar with the application that the data is supposed to model, the user community itself can serve as a validation mechanism by calling attention to anomalous results.

1.2.2.3 Program Validation

The validation of programs in realistic applications environments is usually done by comparing the output from sets of test data with precomputed results. [Beizer, 1983; Myers, 1979] Another approach is to apply techniques to create a formal proof that a program is correct. [Anderson, 1979] Closely related to the technique of program-proving is the approach which has concentrated on developing formal specification methods, which will permit designers to precisely delineate a proposed program's behaviour prior to its creation. [Hoare, 1984] However, neither program proving nor formal specification methods have, as yet, been widely applicable to programs and systems of programs of realistic size. For very large systems, even the more pragmatic test data method may not be adequate to test all realistic possibilities. This problem has stirred research into programming language design, and programming practice studies, whose aim is to facilitate the writing of programs which are easy to understand and thus, presumably, less prone to logical errors. [Green, 1980] Current work in this whole area is concentrated on large-scale software systems, whose very size and complexity introduce problems which are not present in small programs.

1.2.2.4 Validation in the data base context

Those who employ a query language face the problem of the validation of their query language expressions just as professional programmers face the problem of the validation of their programs. Query language users have both advantages and disadvantages as compared to programmers.

On the one hand, an *ad hoc* query expression will not be, in the nature of things, subject to the kind of testing and checking that can be applied to programs of professional programmers. It is unlikely to be run with sets of test data. Anomalous results from a wrongly-constructed query will not necessarily be recognised as such.

On the other hand, a query language is a simpler construct than a fully general programming language. The relational algebra, for instance, has no transfers of control and no recursion. It has only one data structure, the relation. In the query context,these are constructed entirely from pre-defined relations. Queries in the algebra can always be constructed as a series of single operations on singlets or pairs of relations. This simplicity makes possible a high degree of syntactic and semantic error detection, to trap queries which cannot be valid. Of greater difficulty is the task of validating queries which may or may not be valid, according to the user's intentions. As one researcher has commented,

> Since syntactic errors will probably be caught by the system, semantic bugs are the main object of study...the main problem in database access debugging will be the determination that a semantic error has occurred: [since] syntactically correct functions will produce a reasonable event, how are users to know that there is a bug? [Shneiderman, 1978b]

1.2.3 Query Validation Techniques

1.2.3.1 Syntax checking

A query language expression which does not conform to the structural constraints of the query language processor cannot be processed at all, and the invalidity of the query will be obvious.

1.2.3.2 Data base object existence and type checking

A query which makes reference to non-existent data base objects is certainly invalid if those objects are relation or attribute names, and may be invalid if those objects are values. Where a domain has a limited number of values a query which makes reference to that domain can be scanned before processing , and the non-existence of the value referred to (as a necessary and not a contingent fact) can be indicated to the user.

A system which has more than one data type can flag data-type errors. Where constraints such as minimum and maximum values exist on data, constraint-breaking queries can be signalled.

One approach of interest has been to design a "co-operative" query system which spots user misconceptions. For example, where a query would result in a necessarily null result, the system explains why this is so. [Janas, 1979; Kaplan, 1982]

Previous query language studies uniformly report poor spelling by users as a major cause of errors, suggesting that the incorporation of spelling correctors will be standard in future systems which must deal with non-professional users. [Durham, Lamb and Saxe, 1983; Peterson, 1980] Related to simple misspelling errors are synonym errors, in which users give an alternate, synonymous, name to a data base function, relation, or attribute.

A query expression which conforms to the syntax of the query language and the semantic constraints of the data base will execute and produce results. This does not guarantee that the expression is the embodiment of the query the user wants to ask. Additional techniques can be marshalled to allow the user to check the conformity of the query language expression to the user's intentions.

1.2.3.3 Display of execution tree

Some queries can only be satisfied by a group of operations, whether formulated as a single complex expression or as a series of unary or binary operations with explicitly-named intermediate relations. It may be conjectured that the sub-set of such queries which include some operations which can be carried out in parallel with each other will be more difficult to comprehend than those in which the chain of operations has a definite linear ordering. A graphical display of the actual tree of the order of execution could be useful in making the logic of the query explicit. Semantically-equivalent alternate execution trees (such as are generated by a query optimiser) might also prove of value in demonstrating to a user the semantics of a query expression.

1.2.3.4 Run time analysis of the executing query

As a query is executed, the cardinality of the intermediate relations may convey information to the user, who will possibly have some rough idea of the cardinality to be expected from them. Values which are orders of magnitude larger or smaller than expected would signal possible faulty logic in the formulation of the query. (An evaluation of the PRTV system reported that its users found the ability to see the cardinality of intermediate relations "particularly useful" [Storey, 1979]).

1.3. User interfaces with data bases

1.3.1 Natural language communication with a data base

One approach to minimising incorrect queries is to bypass the necessity for the user to learn a conventional query language by giving the computer the ability to understand the user's language. Proponents of this approach argue that a typical user "has only ... [a] vague understanding of the actual structure of the data base" and that we cannot "expect him to express his queries in a langage that requires knowledge of how the data is actually structured" [Harris, 1977] The attempt to develop "natural language front ends" to data base systems was begun in the mid-1960s, and is currently the focus of renewed research and development [Blanning, 1984; Bolc, 1980].

The ability of a computer to deal usefully with a realistic range of natural language input is closely related to its ability to store information about the world to which that input refers. This has meant that, so far, natural language systems have been able to accept input about very limited domains only. [Rich, 1984]. However, precisely because data base applications can indeed be about limited domains, there have been successful, albeit restricted, applications of natural language processing in this field. Most natural language research relevant to data base studies has centered on systems which include a deduction capability as a central feature. A deductive system allows the user to store information of the kind "Every X is a Y", "A is an X", and then ask "Is A a Y?" without having to store the fact that "A is a Y" explicitly. Davey surveys a number of these systems and discusses their limitations. [Davey, 1978] Chang and Lee provide an introduction to the field of mechanical theorem proving upon which "deductive" query systems are built, and classify the kinds of queries such systems are designed to answer. [Chang, 1973] Although we may expect future data base systems, particularly those with natural language interfaces, to include a deductive component, most current practical systems leave deduction to the user. (The deduction may be embedded in the process of query language expression formulation.)

For casual users, whose access to the data base is infrequent and who have a low tolerance for difficulties in using a database interface, some form of natural language dialogue system is indicated. Pioneering work in this field was begun by E.F. Codd, who notes

> To have any hope of being viable, a natural language query system must "talk back" to the user using the same natural language he uses, except that its style has to be much more precise than that of the average user. Accordingly, there must be a natural language generator as well as a natural language analyzer in a viable natural language query system....
>
> Incidentally, very little work on the generation of natural language has been published...and none of this appears to treat the problem in the context of query formulation. [Codd, 1978]

The question must be asked, if dialogue-based natural language systems are sucessfully developed, will they not render obsolete all other query language systems? That the answer is no is suggested by two observations about such systems:

(1) the computer resources required by natural language analyzers will confine their implementation to a restricted set of systems for some time to come. Natural language interpretation by computer requires representation, storage and rapid access to real-world knowledge by the interpreting program, in order to supplement syntactical analysis with semantic analysis. So far this has only been practical where the world to which the natural language conversation refers is a very limited one. Certain data base applications do fit this requirement but the rather limited practical application of such systems to date suggests that we are far from seeing them replace formal query languages.

(2) the dialogue mode, necessary because of the inherent ambiguity of natural language, may be bothersome to a user who is capable of formulating queries in a formal query language. We include "direct end users," in the CODASYL EUFC sense, in this category. As M. Zloof has observed:

> ...there is no <u>one</u> language which satisfies the needs of the entire spectrum of potential users, but rather classes of languages suitable to classes of users. We start by dividing the potential user community into three fuzzy categories:
>
> 1. Casual users;
> 2. Non-programmer professionals;
> 3. Application programmers.
>
> ... A non-programmer professional...is a person motivated by the job and familiar with the particular application. One can expect such a

> user to learn an easy-to-use formal language and be familiar with the concepts normally required by a relational model. In this category one can include secretaries, clerks, engineers, analysts, etc. [Zloof, 1978]

This view is in contrast to the categorical exclusion of formal query languages as end-user interfaces expressed by Harris above.

1.3.2 Pseudo-natural language systems

Many query languages, including several designed for a relational data base, use English keywords to achieve a superficially English-like appearance. Whether this is an improvement over a purely symbolic language is an open question. As one researcher notes:

> The pseudo-English used in query systems has often been more unnatural than computer languages themselves...[and] The actual effectiveness of pseudo-natural language systems has to be carefully evaluated, since the examples presented in the literature are generally chosen to convey the strength rather than the weakness of the given approach. [Wiederhold, 1977]

Another potential difficulty with pseudo-natural language systems was highlighted by a study made by three researchers investigating programming errors. [Green, 1980] They reported the results of an experiment to test the design of conditional statements, comparing user' errors made with a language requiring ALGOL 60-like conditionals (IF B1 T@EN BEGIN S1;S2 END ELSE BEGIN S3 END) against those made with a language permitting a more "natural" structure (IF B1 THEN S1; S2 ELSE S3 END). This experiment found what the experimenters expected to find with the first language: syntax errors and consequent debugging difficulty due to failure to match BEGINs and ENDs. However, to their surprise, the experimenters found that almost none of their subjects was able to use the second, "natural", language to program a solution that required the nesting of conditionals, even though the instructions had included worked examples of just such problems. As they report of their subjects' problems

> The clue to their strange inability seems to be given by their frequent complaint, when shown the solution, "But that's not English!" What seems to have happened is that subjects learning language 1 could see quite readily that the language was not English, so they were forced to read the instructions we gave them. Subjects learning language 2, however, decided that it was so like English that it was all quite obvious and natural, so they only needed to skim through the instructions, only to come to grief when they met a problem requiring the un-English construction,
>
> ```
>      IF green THEN
>          IF juicy THEN ...
> ```
>
> ...we seem to have an interesting suggestion that making programming languages look like natural languages may actually make them harder...

The last few years have seen rapid growth in the use of microcomputers with sufficient power to sustain query interfaces with a degree of (claimed) natural language capacity. The market is now providing both the impetus for the development of natural language interfaces and a test of their perceived usefulness. Thus the debates cited above may relatively soon receive an empirical judgement.

1.3.3 Query validity's relationship to data models and language choice

1.3.3.1 Query validity in the context of data models

Considered from the viewpoint of their predisposition towards valid use, query languages whose expressions must be formulated in the context of a complex data structure would intuitively seem to be at a disadvantage when compared to query languages whose arguments are simpler data structures. This argument was often advanced by advocates of the relational approach, and the assertion subjected to several attempts to test it empirically. [Date and Codd, 1974; Greenblatt and Waxman, 1978; Kuhn and Shneiderman, 1978; Lochovsky, 1978; Reisner, 1977]

Interest in, and theoretical elaboration of, the relational approach occurred just at the beginning of the era of, and was partially motivated by, the expansion of the number of end users of data base systems which has marked the last decade. There is an element of paradox in the fact that, although one of the main arguments for the relational system is its conceptual simplicity, just this very simplicity can give rise to new difficulties in the field of query validation. As one researcher has commented:

> Information obtained from traditional data processing systems regarding typical queries cannot be extrapolated to relational data base systems since the greater flexibility of the relational systems promotes more complex queries. No real experience of relational data base

systems over a wide variety of applications has been obtained. [Hall, 1975]

1.3.3.2 Query validity and query language type

Relational data bases can sustain a variety of query langages, including the types previously discussed. These languages are usually elaborations of one of the two approaches first delineated by E.F. Codd, and called by him "relational algebra" and "relational calculus". The distinction between the two lies in their respective procedurality: the "algebra" defines the resultant relation by explicitly specifying the operations to be done on constituent relations to produce it, where the "calculus" is simply an expression specifying the resultant relation. Query languages have been designed which are more or less direct implementations of the algebra. Calculus-based systems are more usually embedded in a front end which shields the end user from the somewhat daunting predicate calculus-style expressions. The power of the two approaches is equivalent.

1.3.3.3 Suitability of Relational Algebra as a Query Language

It should not be assumed that professional non-programmers will be unable or unwilling to learn a simple, albeit mathematically-flavoured, query language. Here experience with the programming language APL may be relevant. Despite its lack of control structures, its cryptic error messages and its limitation to a single general data type, the ease of use and simple syntax of this language has allowed it to be taken up by a substantial group of users falling into the "professional non-programmer" class, including not only engineers but also managers. The relational algebra is qualitatively more simple than APL and is thus not an unrealistic choice as a language for end-user oriented research.

One experiment which tested user performance using an algebraic language, a keyword-oriented "mapping" language (SEQUEL) and the forms-oriented Query-By-Example found better performance with Query-By-Example, whose users got 75% of their queries correct, as against 73% for the SEQUEL users and 67% for the algebra users. [Greenblatt and Waxman, 1978] However, as noted by another researcher in this field, the great variability among the groups of test subjects raises doubts about the applicability of the results claimed. [Shneiderman, 1978] (For instance, the algebra users were almost evenly divided between males and females, while the Query-By-Example users were three-quarters male.) Another criticism of this experiment is that the algebraic language used, SQUARE, has a rather opaque syntax: there are no distinct operators for each operation, but rather the location of the relation names and the type of brackets determines the kind of operation to be carried out. (In contrast, the ISBL relational algebra used to illustrate this thesis has a distinct operator for each operation.)

1.3.4 A Review of User Performance Studies

In "The Use of Psychological Experimentation as an Aid to the Development of a Query Language" P. Reisner presents a number of conclusions and further hypotheses generated by an experiment testing the usability of two relational query languages, SQUARE and SEQUEL. [Reisner, 1977] Five types of "minor error" were identified:

(1) Ending Errors: users often employed "syntactic variants" of data base object names and keywords, such as "supplied" for "supplier", or "name" for "names".

(2) Straightforward spelling mistakes.

(3) Synonym errors: "AVR" for "AVG".

(4) Quotation Mark errors: the version of SEQUEL used in this experiment requires users to omit quotation marks from numeric data values, while requiring them around string values, a common programming language convention. However, users evidently found this "rule and exception" more difficult to apply than a simple requirement to enclose all data values of whatever type in quotation marks.

(5) Punctuation errors: omission of periods, commas, colons, etc. Reisner noted that "the high frequency of these came as a surprise," and recommended

that future systems include a spelling corrector, a synonym dictionary and a stem-matching procedure, perhaps in the form of an interactive dialogue interface.

Major errors were classified as "format errors", which would prevent the query from compiling, and "substance errors" which would run but produce a wrong answer. The latter were judged to be less serious than the former, since a format error showed that a user did not understand the syntax of the language at all. While this judgement is understandable in a teaching situation, in actual use the more serious error, in terms of its consequences, would be a "substance error", since it can go undetected. (A "format error" signals its presence unmistakably.)

One of the more interesting results of this experiment, bearing directly on the work of this thesis, was the remarkable consistency with which users made "substance errors". The test population was divided into four groups: non-programmers using SQUARE, non-programmers using SEQUEL, programmers using SQUARE and programmers using SEQUEL. Great variation was exhibited between programmers as a group and non-programmers (78% of programmers' queries were essentially correct, as against 60% of non-programmers). Choice of language, while of little importance for programmers, affected non-programmers performance significantly (using SQUARE, non-programmers got only 43% of their queries correct; with SEQUEL, the percentage correct increased to 56.)

Showing little variation, however, was the percentage of queries called by the experimenters "substance errors," which produced an answer, but the wrong one. About one out of ten queries put by each group fell into this category, with both programmers and non-programmers, SEQUEL users and SQUARE users, erring with the same frequency. (The range was from 11.5% to 10.2%.)

If these results are indicative of what may be expected when casual users have access to a data base via a query language, the importance of developing methods to minimise the occurrence tf "subs
ance errors" is clear.

User errors were also classified by the experimenters according to their probable causes. Using this scheme, the following six types of causes of error were identified:

(1) Intrusion errors: these were errors "directly traceable to the wording of the English question." An example was a question asking for the "names" of personnel, held in the data base under the column NAME. Many users formulated their query using the term NAMES, evidently carrying the English

language phrase directly into the query.

(2) Omission errors: where the English sentence asked for information that required certain query language expressions, but contained no direct "clue" signalling that the particular expression was necessary. An example was the question "How many 747s are dispatched from LaGuardia Airport?" The SEQUEL expression required was

```
SELECT NUMBER-DISPATCHED
FROM DISPATCH
WHERE AIRPORT = "LAGUARDIA"
AND   PLANE   =  747
```

The experimeters conjectured that leaving out the word "plane" from the English expression caused some subjects to miss out the last clause in the formal query.

(3) Prior-knowledge errors: evidently, some subjects already "knew" certain computer-communication conventions which happened not to be the conventions being used by the query system. An example was using alternative abbreviations for common system functions, such as AVR for AVG.

(4) Domain incompatibility errors: where two columns appeared to hold values from the same underlying domain, but did not, subjects sometimes formed queries as if they did, being misled by similarity of attribute names. An example would be the attempt to intersect airport names with person names, each (incompatible) set being represented by a

column labeled NAME.

(5) Consistency errors: these were described as those arising from the assumption by subjects that the query language used was strictly consistent. An example would be the expectation that quote marks must be applied to all constants, regardless of their data type. The experimenters conjectured that this was "simply a preference for one rule over two."

(6) Overgeneralisation errors: Some errors seemed to be caused when "subjects appeared to develop rules for query writing (not necessarily consciously), and then used these rules in inappropriate places." An example would be using the table DISPATCH successfully to answer a query about "planes dispatched", and (falsely) assuming that "dispatched" in English is always rendered as "dispatch" in the query language, when in fact the system includes such column titles as NUMBER-DISPATCHED.

In this paper Reisner proposed a preliminary measure of query complexity, based on the number of "transformations" necessary to turn a natural language expression into a formal query language expression. This approach assumes that a "query" is identical to a particular natural language expression embodying it. It thus would be reasonable to assume that the more complex and difficult to comprehend the natural language expression is, the more errors are likely to be made by a user attempting to translate the natural language expression into a formal language. For example, if the user's query is "how many jumbo jets fly from Heathrow daily?" and the existing data base records information on "747s" instead of jumbo jets, then a transformation is needed at this point from "747" to "jumbo jet".

The above assumption may be criticised on the following grounds: many users do not begin with someone else"s natural language formulation of a query, but with an <u>abstract query</u>, which can take more than one natural language form. In the next section we discuss the concept of the abstract query.

1.4 <u>Classification of Queries</u>

1.4.1 <u>The Abstract Query and its notations</u>

The term "query" can have at least three distinct, although related, meanings when used to refer to manipulation of a data base. Although the meaning of the term in this thesis will usually be clear in context, a systematic exposition of its meanings and their implications is a necessary preface to further discussion.

In <u>The Logic of Questions and Answers</u> Belnap and Steel develop an approach which they name an "eroteric logic" to capture the grammar and semantics of questions and answers. [Belnap and Steel, 1976] Although their work was "guided by the potential applicability of eroteric logic to current problems in data processing" it is at too high a level of abstraction to be directly applicable to current data base query studies. Nevertheless, a number of their insights and characterisations of questions illuminate the process of query specification. Particularly useful is their insistence that

> The meaning of a question addressed to a query system is not to be identified with how the system processes the query (and is not to be identified with a program at any level), but rather it is to be identified with the range of answers that the question permits. That is, for a query system and a user to agree on the meaning of a question is for there to be agreement as to what counts as an answer to the question, regardless of how, or if, any answer is produced. This conceptual feature is important because only if one has an analysis of questions that is independent of computers and programs can he sensibly ask such questions as these: What sorts of questions would I really like to ask? For various sorts of questions is my query system able to answer them? (Is it "complete" in these respects?)

We shall make use of their distinction between a question as an abstract object, and the notation which may be employed to represent that object (called by them an "interrogative"). We distinguish between the abstract query, and two forms of notation for it: its expression(s) in natural language, and its expression(s) in a query language. In general, there may be many equivalent forms of each of these two notations.

As an example, a particular abstract query may be expressed in the following equivalent natural language notations:

> Find all salesmen who earn more than £10,000/year.
>
> List every salesman with an annual salary greater than £10,000.
>
> Find each employee who makes better than £10K and who is a salesman.

Of course these notations are only equivalent if certain assumptions are true. (For instance, that annual earnings are the same as annual salary.) Such assumptions are not marginal to the process of query formulation, but involve the way in which the data base view of the world being modelled has been organised. Allowing the user to comprehend this view -- to know the data base definitions, categories, business rules, relationships, constraints -- is critical to allowing him to formulate queries correctly.

The same abstract query, some of whose natural language representations are illustrated above will also, in general, have multiple possible correct query language notations. In fact, it will have "families" of notations, one family for each alternate schema of the data base affecting the objects referred to in the query. Thus if the categorisation of employees into employee-types (salesmen, clerks, production workers) is carried out by having a separate relation for each type, the following queries would be equivalent in the ISBL Relational Algebra:

```
LIST SALESMEN:[SALARY > 10000]%[E-NAME]
LIST SALESMEN%[E-NAME,SALARY]:[NOT(SALARY<=10000)])%[E-NAME]
```

A schema which grouped together all employees into a single relation, representing their type by an attribute value, would support the following equivalent queries:

```
LIST EMPLOYEES:[TYPE="SALESMAN" AND SALARY>10000]%[E-NAME]

LIST (EMPLOYEES:[TYPE="SALESMAN"]%[E-NAME]) ^
     (EMPLOYEES:[SALARY > 10000 ]%[E-NAME])
```

In this thesis we shall, therefore, distinguish between

(1) <u>the abstract query</u>, for which there is no single representation,

(2) <u>the abstract query expressed in a natural language notation</u>, noting that there will in general be many possible equivalent natural language notations for a given abstract query, and

(3) <u>the abstract query expressed as a query language expression</u>, with the latter having many possible equivalent forms.

1.4.2 Meaningful and Meaningless Queries

Our concern is to assist the user in mapping an abstract query to a valid query language expression. Before we are able to engage this question, it is necessary to delineate the boundaries around the class of possible abstract queries which can in fact be considered. Clarification is again available from the work of Belnap and Steel.

In the terminology of Belnap and Steel, data base queries are <u>elementary which questions</u>. These are questions to which there corresponds a set of statements which are directly responsive to the question asked. What counts as an answer to this sort of question must be well-defined. (They contrast such questions to "problem-solving situations" and "please relieve my vague puzzlement situations". An example of the latter would be "Who is that man living next door?", since it is not clear what kind of answer would satisfy the questioner: a proper name, some sort of description, or what?) Which-questions implicitly present a set of alternatives, one or more of which is the answer to the question.

In the context of this thesis, a meaningful abstract query is one that can be cast into at least one semantically and syntactically valid query language notation (in the sense of section 1.2.3.2 in this chapter) such that the response to that query can in principle be judged true or false. A query language expression which elicits a false response from a data base may still be a valid expression of a meaningful query, since the software which implements the query language may have faults, or incorrect data may be present in the data base. We have asserted that it is wrong to identify an abstract query with any string of symbols. Statements about the "meaning" of an abstract query are statements about the truth or falsity of the response to that query were it to be cast in the form of a query language expression and put to a data base. (For an analogous approach to the question of natural language semantics, see [Sampson, 1975].) From this it follows that assertions about the meaningfulness of an abstract query can only be made in the context of a particular query language system and a particular data base schema.

Given a relational database whose domains are entity-identifiers for persons and entity-identifiers for languages, and for which the query language is a pure relational algebra, queries about aircraft schedules are meaningless (because the relevant entities are not represented in the data base), and queries about number of persons represented in the data base are meaningless (because the only quantifiers sustained by the pure algebra are those of first-order predicate calculus: "some", "all" or "none"). An abstract query which specifies the retrieval of data, any part of which refers to non-existent data base objects, or which requires a non-existent query language function, is meaningless. (Note that an abstract query may be meaningful, but its query language expression be invalid, due to user misspellings, misconception of the data base schema, and so forth. This is not the same thing.) This thesis deals with ensuring valid query language expressions for meaningful abstract queries.

1.4.3 Other Approaches to Query Classification

James Martin defined a simple query as one that may be made on a single file (or relation), considered to hold information on occurences of entities (E) in the form of values (V) formatted as attributes (A). [Martin, 1977] Letting A(E)=V symbolise the statement that attribute A of entity E has value(s) V. Six permutations of query types that result from replacing one, and then two, of the known values with query-indicators are shown in Figure 1.1 for the relation shown in Table 1.1.

S

SNum [SALESMEN]	Mon [MONTHS]	Income [MONEY]
23	Jan	850
256	Jan	455
271	Jan	970
.	.	.
.	.	.
.	.	.
23	Feb	670
256	Feb	480
.	.	.
.	.	.
.	.	.

Table 1.1 From Martin [1977].

1. A(E)=? What is the value of attribute A of entity E?
How much did salesman 271 earn in January?

```
((S:[SNum=271]):[Mon="Jan"])%[Income]
```

2. A(?)=V What entity E as a value of attribute A equal to V?
What salesmen earned more than £1000 in January?

```
((S:[Income>1000]):[Mon="Jan"])%[Snum]
```

3. ?(E)=V Which attribute or attributes of entity E have value V?
In which months did salesman 271 earn more than £1000?

```
((S:[Income>1000]):[Snum=271])%[Mon]
```

4. ?(E)=? What are the values of all attributes of entity E?
What are salesman 271's earnings for each month?

```
S:[Snum=271]%[Mon, Income]
```

5. A(?)=? What are the values of attribute A for all entities?
What did all salesmen earn in January?

```
S:[Mon="Jan"]%[SNum, Income]
```

6. ?(?)=V What are the attributes of all entities having value V?
List the salesmen who earned more than £1000 in any month, and the month itself.

```
S:[Income>1000]%[SNum, Mon]
```

Figure 1.1: <u>J. Martin's Classification of Query Types</u>

Lacroix and Pirotte defined a query with respect to the "objects" and "properties" of the data retrieved as a result of the query:

> Very generally, a query requests one or several sets of objects which satisfy certain properties; these properties are in turn expressed with the help of other objects and properties.... In general, properties make reference to data base relations and objects of other domains. [Lacroix and Pirotte, 1977]

Sundgren distinguished between queries about the world being modelled in the data base, and queries about the data base itself, while noting that there need be no structural difference between these two types of query if their respective data bases are modeled on identical principles. [Sundgren, 1975] (In relational terms, this would require that the data dictionary be itself a set of relations.) He categorised queries along a second axis, dividing them into yes-no type queries, retrieval queries, and process queries. By the latter he meant queries in which

> The operator or the parameter part will contain a <u>processing request</u>, meaning that not only should a specified set of messages be retrieved, they would also be processed in a certain way before presentation. For instance, the processing request could imply aggregation and statistical analysis.

He noted that retrieval queries are the most fundamental type of query, inasmuch as most process queries will require an initial retrieval and yes-no queries may be considered a special case of retrieval queries. He observed that

> Many retrieval queries conform to the pattern, For objects having the property P, retrieve the

values of the attributes A1,...,Am...

Sundgren drew a distinction between attributes involved in specifying the property P, so-called "alpha" attributes, and attributes to be displayed, the "beta" attributes. An example of both sorts of attributes would be the request to "PRINT the SNum [a beta attribute] of each salesman with Income > 850 [an alpha attribute].

A similar definition of a query is made by Ghosh , who described a query as having two parts,

> One is called the <u>qualification part</u> and the other the <u>target part</u>. The qualification part specifies the properties that have to be satisfied by an individual of the universe in order that it may be a relevant piece of information for the query. The target part of the query specifies what information about the individuals, who are relevant to a query, are [<u>sic</u>] needed for the answer. [Ghosh, 1977]

It is interesting to note that almost all of the above authors approach the defining of queries by assuming that the target of a valid query is a set of nameable objects/entities/individuals which is to be retrieved. (The exception is James Martin, whose queries 5 and 6 call for the retrieval of tuples.) The general assumption is that it is a single column that will remain, a relation of degree one. But we often want to retrieve more complex tuples: for instance, persons, and the languages they speak. Not two relations, one holding persons and the other languages, but the person-language pairs. Outside of the data processing context, there is no word for such associations. [Kent, 1978] (Within the data processing context, we might call associations "records" or "tuples".) This lack of vocabulary for many kinds of associations implies that there is no "natural" mental concept for them. This in turn suggests that perhaps it would be easy to formulate queries where the objects used in associations were either single entity sets, or tuples which were intuitively one entity and a group of attributes -- and hard to formulate queries where genuine relationships among entities needed to be retrieved. This possible problem is relevant to one of the design choices made in designing the system described in this thesis, where the problem of articulating complex relationships among the objects described by a tuple is central.

Using the terminology of the authors just cited, we shall assume that it is in the specification of the _properties_ or "_alpha attributes_" or _qualification parts_ that the greatest difficulty lies for end users.

1.5 Review of work related to Reverse Translation

1.5.1 Natural Language Generation: Background

The algorithmic generation of grammatical sentences in natural language -- in contrast to its inverse, the algorithmic classification of any candidate natural language word-string as grammatical or not -- is a trivial matter, provided that two conditions are met:

First, that a sub-set of all possible grammatical sentence-forms is acceptable (since a complete grammar of a natural language involves thousands of rules). Second, that the definition of "grammar" is restricted to syntax rules only (permitting, for instance, the generation of sentences such as "colourless green ideas sleep furiously"). Given these limitations, a simple phrase-structure grammar implemented as a set of production rules and drawing on stocks of words as its terminal strings can generate natural language sentences. (There are also numerous other grammatical formalisms available for language generation, although most of these have been developed as part of research into natural language understanding.)

Of interest in the database query context are those natural language generation systems which have as their purpose the conveying of information about some computationally-tractabledomain of information held in machine-accessible form.

In such applications, the theoretically challenging aspect of natural language generation consists in defining the method of mapping between the domain about which one wishes to convey information, on the one hand, and a set of natural language sentences conveying the desired information, on the other.

1.5.2 Natural Language Generation: Non-Database Examples

Davey developed a computer program that produced English discourse, using a systemic grammar. [Davey, 1978] The program was described as "capable of describing in a sequence of English sentences any game of noughts and crosses (tic-tac-toe), whether given or actually played with the program." It was written in POP-2, and took between thirty seconds and three minutes to produce each sentence. The intension of the author was to model the way a human speaker chooses to present information in sentences, taking into account what has already been said and what the hearer can be expected to know.

As part of a user-friendly command interface, the UNCLE system generates natural language explanations of exception conditions which can occur when users make errors in operating system commands. [Efe, Hopper and Miller, 1983] The system analyzes operating system messages and maps them onto a meaning-representation system. This system consists of five "paradigms", which are supposed to be used by human beings when they speak, each one of which represents a different sort of entity relation. When one of these paradigms can be filled in by the message analyzer, and combined with a particular case structure, it corresponds to a particular phrase-structure grammar production rule. This rule can then be invoked to generate a sentence which is semantically equivalent to the operating system message, but which uses the terminology, concepts, and assumptions of the particular user for whom the system is tailored.

The UC natural language help facility for the UNIX operating system includes a language generator, although at the time of the most recent report on the system its designers reported that it was "quite sketchy, largely because most of our effort has gone into request understanding rather than answer generation." [Arens, Chin and Wilensky, 1984]

1.5.3 Natural Language Generation: a Database Example

James Longstaff and colleagues produced a query system which generated natural language sentences corresponding to partial queries put in a relational calculus. [Longstaff, Poole and Roper, 1978] In response to an incomplete query calling for the retrieval of some of the attributes of a particular entity, the system would offer the user the chance to select either a universal or existential quantifier for each of the other entities to which the target entity was related.

Given a database with a schema consisting of three "entity relations" (SUPPLIER, with attributes SNO, SLOC, SNAME; PROJECT, with attributes JNO, JNAME, and JLOC; and PART, with attributes PNO and PTYPE) and one "relationship relation" (SUPPLY, with attributes SNO, JNO, PNO, and DR), a partial query put by a user might be

```
SELECT SUPPLIER [SNAME, SLOC]
WHERE  PART [PTYPE] = A  AND PROJECT [JLOC] = SJ
```

The system would respond,

> Select names and locations of suppliers where
> each supplier has supplied (all/a) part(s) of
> Type A to (all/a) project(s) located in SJ.

(The quantifiers enclosed in parentheses represent choices to be made by the user.)

It is not clear from the cited paper how the system as described would cope with negations, or with schemas where the same set of entity sets were related in more than one way (for instance, with an additional three-part relation, PROMISED, having the same attributes as SUPPLY).

The method utilised to generate natural language sentences depends critically on the fact that the so-called "relational calculus" query expression is a description of the final relation desired by the user, as opposed to a specification of the procedures (relations and operations upon them) which should generate that relation. This work thus has no directly-transferable techniques which could be applied to a natural language generation system for a "relational algebra" query processor, where a query is put by the user explicitly specifying the relations and operations upon them which should yield the desired result. However, although the system described here was not developed further, it must be acknowledged as the first developed "Reverse Translation" approach to query validation.

CHAPTER TWO

FOUNDATIONS OF REVERSE TRANSLATION

2.1 Introduction

This chapter introduces and defines the terms which will be used in the remainder of the thesis, and illustrates the relational algebra which will be employed in demonstrating reverse translation.

2.2 Data Models

2.2.1 The Relational Model

2.2.1.1 Definitions

Given the domains, not necessarily distinct, D1, D2,...,Dn, a relation on these domains is a set of tuples where for each tuple its first element is a member of D1, its second a member of D2 ... and its Nth a member of Dn. The value of "n" is in relational terminology the degree of the relation. (For reasons to be made clear, we shall henceforth use the term "Rdegree" for "degree" in the sense just defined.) The number of distinct tuples is the relation's cardinality. The i-th element of each tuple constitutes an attribute of the relation: attribute names must be distinct within a relation. Relation names must be distinct within a data base. A set of attributes which uniquely defines a tuple is a key of the relation.

As an example, consider the many:many relation PSL (Table 2.1) which records persons and the languages spoken by each person. (This example also displays the conventions which will be followed henceforth in displaying examples of relations, their names, domains, attributes and tuple values. Relation names will be displayed in upper-case type, attribute names will be underlined, and domain names will be in upper case, appearing beneath attribute names and enclosed in square brackets.)

Relation Name ------>	PSL	
Attribute Names ----->	P	L
Key ----->	P,L	
Domain Names ----->	[PERSONS]	[LANGUAGES]
	Adam	English
	Adam	French
	Gunther	German
	Jean	French
	Uli	English
	Uli	French
	Uli	German
	Zahid	English

Table 2.1able 2.1. The relation PSL.

2.2.1.2 The Relational Algebra

A set of relational algebra operators can be defined which when combined with relations in expressions yield relations as results. The following algebraic operators constitute a formal query language which has the power of the first-order predicate calculus. (Actual implementations which have used a relational algebra as a query language have supplemented it with additional functions. The following examples demonstrate the syntax of each relational operation which has been given a Reverse Translation extension. The syntax and conventions of the query language thus defined are based closely on, but not identical to, the Information System Base Language (ISBL) of the Peterlee Relational Test Vehicle. [Todd, 1976] They differ from ISBL in the following ways:

(1) The DIVISION operator, used for formulating queries with universal quantification, is not included. DIVISION is in fact a redundant operator, if DAFFERENCE, PROJECTION and JOIN (Cartesian Product) are available. The problem of Reverse Translation of queries involving universal quantification is taken up in Chapter Eight.

(2) PERMUTATION of a relation is handled in ISBL via a PROJECTION operation in which no attributes are dropped. Here, for clarity, it has been given a special operator of its own.

(3) RENAME of attributes, necessary to permit the set operations and join, has also been given a special

operator, again for clarity. In ISBL its syntax was part of the definition of the PROJECTION operation.

(4) SELECTION in ISBL includes the possibility of complex expressions which include logical operators (AND, OR and NOT). Any relation yielded by such an expression can also be derived by an equivalent series of SELECTION, INTERSECTION, UNION and DIFFERENCE operations. In the interests of simplicity, the SELECTION demonstrated in this thesis will be defined for single comparisons of one attribute to a value from the domain of that attribute. (This also excludes the so-called RESTRICTION comparison between two attributes.)

(5) ISBL permits complex relational expressions, in which th result of one expression could be used as an argument to another. However, to facilitate discussion and illustration of the Reverse Translation method, all query examples in this thesis using more than one operator will be formed as multi-statement queries utilising explicit, named intermediate relations.

: SELECTION   Selection acts on a single relation to produce a second relation which is a sub-set of the first, having only those tuples which are specified in a comparison operation between an attribute and a value from the domain of that attribute.

An example of selection:

P1 <- PSL : [L = "French"]

```
Relation
Name ------>     P1

Attribute
Names ----->      P            L

Key   ----->      P,L

Domain
Names ----->      [PERSONS]     [LANGUAGES]
```

P	L
Adam	French
Jean	French
Uli	French

Table 2.2. The relation P1.

% PROJECTION PROJECTION acts on a single relation to produce a second relation having only those attributes which were named in the PROJECTION operation.

An example of projection:

```
P2 <- P1 %[P]
```

```
Relation
Name ------>      P2

Attribute
Names ----->      P

Key   ----->      P

Domain
Names ----->      [PERSONS]
```

Adam
Jean
Uli

Table 2.3. The relation P2.

Set operations The next three operations take as operands two relations of identical degree whose respective attributes must be drawn from the same domain.

To illustrate the set operations assume a relation P3, formed by selecting all English-speakers from P1 and then projecting on the P attribute.

```
Relation
Name ------>      P3

Attribute
Names ----->       P

Key    ----->      P

Domain
Names ----->       [PERSONS]

                 _________________
                !  Adam           !
                !  Uli            !
                !  Zahid          !
                !_________________!
```

Table 2.4. The relation P3.

U UNION   UNION acts on two relations to produce a third containing all of the tuples appearing in either operand.

An example of UNION:

```
P4 <- P2  U  P3
```

```
Relation
Name ------>      P4

Attribute
Names ----->       P

Key    ----->      P

Domain
Names ----->       [PERSONS]
                 _________________
                !  Adam           !
                !  Jean           !
                !  Uli            !
                !  Zahid          !
                !_________________!
```

Table 2.5. The relation P4.

- <u>DIFFERENCE</u> DIFFERENCE produces a relation whose tuples are those which are in the left-hand operand but not in the right hand operand.

An example of DIFFERENCE:

P5 <- P2 - P3

```
Relation
Name ------>      P5

Attribute
Names ----->      P

Key   ----->      P

Domain
Names ----->      [PERSONS]
```

Jean

Table 2.6. <u>The relation P5</u>.

^ <u>INTERSECTION</u> INTERSECTION produces a relation having only those tuples which are in both the left-hand and right-hand operand relations. INTERSECTION is actually redundant, in two respects:

(1) if DIFFERENCE is defined, INTERSECTION is unnecessary, since A ^ B <=> A - (A - B);

(2) if JOIN is defined, INTERSECTION is unnecessary since it is merely an equi-join over all attributes. However, it is found in most if not all relational query languages and is therefore included for convenience.

An example of INTERSECTION:

P6 <- P2 ^ P3

```
Relation
Name ------>     P6

Attribute
Names ----->      P

Key   ----->      P

Domain
Names ----->      [PERSONS]

               _____________
              !  Adam       !
              !  Uli        !
              !_____________!
```

Table 2.7. The relation P6.

* JOIN JOIN takes as operands two relations and produces a relation which is the concatenation of the operands except for specified attributes which are shared. The tuples of the resultant relation are drawn from those tuples of the operand relation sharing identical values ("equi-join") in the specfied common attributes. The attributes to be shared in each relation are specified by having identical names. If no attributes in each relation have names in common, a full Cartesian product results (each tuple of the first relation is joined to every tuple of the second relation).

To illustrate JOIN, assume we have a relation PRL, recording persons and the languages they read, which we wish to JOIN to the "speakers" relation PSL illustrated in Table 2.1

PRL

P [PERSONS]	L [LANGUAGES]
Adam	English
Gunther	German
Mike	Danish
Uli	English
Uli	French
Uli	German
Zahid	Chinese

Table 2.8. <u>The relation PRL</u>.

```
PSL @ [ L -> LS ]  (Rename the non-participating
PRL @ [ L -> LR ]   attributes.)
PSRL <- PSL * PRL

                     (JOIN  PSL and PRL over the
                     commonly-named attribute, P.)
```

Relation Name ------> PSRL

Attribute Names -----> P LS LR

Key -----> P, LS, LR

Domain Names -----> [PERSONS] [LANGUAGES] [LANGUAGES]

P [PERSONS]	LS [LANGUAGES]	LR [LANGUAGES]
Adam	English	English
Adam	French	English
Gunther	German	German
Uli	English	English
Uli	French	English
Uli	German	English
Uli	English	French
Uli	French	French
Uli	German	French
Uli	English	German
Uli	French	German
Uli	German	German
Zahid	English	Chinese

Table 2.9. The relation PSRL.

PERMUTATION The PERMUTATION of a relation produces a new relation with the same population of tuples, but reordered according to the order specified in the PERMUTATION operation.

```
LSP <- PSL # [ L, P ]

Relation
Name ------>      LSP

Attribute
Names ----->   L     P

Key   ----->      L,P

Domain
Names ----->      [LANGUAGES]   [PERSONS]
```

English	Adam
French	Adam
German	Gunther
French	Jean
English	Uli
French	Uli
German	Uli
English	Zahid

Table 2.10. The relation LSP.

@ Rename Like PERMUTATION, RENAME is not an operator proper but is a necessary auxiliary to permit operations whose syntax expects particular values for attribute names.

```
SSL <- PSL @ [ P -> S]
```

```
Relation
Name ------>       SSL

Attribute
Names ----->    S     L

Key    ----->       S,L

Domain
Names ----->       [PERSONS]      [LANGUAGES]
```

Adam	English
Adam	French
Gunther	German
Jean	French
Uli	English
Uli	French
Uli	German
Zahid	English

Table 2.11. The relation SSL.

2.2.2 The Entity/Relationship Model and its Application in Reverse Translation

Because the relationships between the values in a tuple are critical to understanding the meaning of a relation, constant reference to them will be made in the remainder of this thesis. A reasonably well-understood vocabulary and rudimentary underlying semantic model with which to conduct this discussion is therefore needed, and a form of the Entity-Relationship model to supplement the conceptual tools already defined has been chosen. In the taxonomy of the principal author of the Entity-Relationship approach, we will use a "Generalised Entity-Relationship Model without Attributes" [Chen, 1981].

This thesis will refer frequently to entities and to the relationships which hold among the entity occurences represented in the tuples of a relation which has been created as the result of the execution of a user's query. The terms "entity occurence", and "entity set", will be used as equivalent to "attribute value" and "domain", respectively, in the standard relational terminology. The possibility of distinguishing between an entity occurence and its representative within data base is allowed, at the level of text generation. The term "relationship", with no formal relational equivalent, cannot be dealt with so casually and this concept is thus the object of more extensive treatment in section 3 of this chapter.

Although the Reverse Translation system is designed for translation of relational algebra queries put to a relational database, the relational terminological apparatus alone, even as supplemented by the Entity/Relationship model, is both too abstract and too meagre to provide a sufficient basis for discussion of methods of generating meaningful translations of relational query language expressions. For example, as many others have pointed out, there is no place in the pure relational formalism for inclusion of information about the kind of relationship obtaining among the objects which make up a tuple of a relation. [Schmidt and Swenson, 1975; Sowa, 1975]

In the relation (Table 2.1) used to provide examples of the conventions used in this thesis, it will be necessary for the user of the database to know that the relationship recorded here is one of "speaks" (in the PERSON to LANGUAGE direction) and "is spoken by" in the LANGUAGE to PERSON direction. Since this information is not part of the relational apparatus, if it is to be used as a component of a Reverse Translation facility, it must be added as part of an extension made to the relational system. (In an unextended system, this sort of information may be held as unformatted descriptive text associated with a particular relation, or deduced by the user through applying his world-knowledge to the names given to the relation, its domains and its attributes.)

The subject of developing a data model powerful enough to incorporate in a natural manner real-world semantics of interest to users is one which is advancing rapidly. The data model assumed in this thesis is intensionally one which is shorn of the refinements which are possible, in order to concentrate on a single aspect of user interface design.

2.2.3 Reduced Relations and Normal Forms

A relation embodies relationships among entities. The simplest embodiment of a relationship in a relation is one n-ary relationship per relation, and one relationship per n-ary relation. But perfectly valid relations may exist which complicate matters in two (not mutually exclusive) ways:

A relation could hold multiple relationships among the entity-identifiers of a given tuple. For example, the relation PRLA which recordswhich persons read which languages, and which languages use which alphabets.

PRLA

P [PERSONS]	L [LANGUAGES]	A [ALPHABETS]
Adam	French	Latin
Adam	German	Latin
Gunther	English	Latin
Gunther	German	Latin
Ivan	Farsi	Arabic
Ivan	French	Latin
Ivan	Russian	Cyrillic
Mehmet	Turkish	Arabic
Mehmet	Turkish	Latin

Table 2.8. The relation PRLA

In this case, the requirement that all relations be held in Third Normal Form would eliminate examples of the above type. But the Third Normal Form requirement is not sufficiently powerful to ensure the degree of simplicity in the relational schema required for the implementation of Reverse Translation. This is because a relation in Third Normal Form can still represent multiple relationships between the set of attributes making up the key, on the one hand, and any number of non-key attributes each of which is uniquely determined by the key. (An example would be a relation in which the first attribute is a unique identifier for a person, the second is that person's birthdate, the third is the name of the country in which the person was born, the fourth is the person's height, and so on.)

2.2.4 Relational Families

Any relation of relational degree N (N > 1) which has an attribute consisting of M entity identifiers can be decomposed into M constituent relations of relational degree N-1, by creating a new relation for each distinct value of the "absorbed" attribute. For example, instead of relation PSL, we could have relationsPSLF(everyone whospeaksFrench), PSLG (the German-speakers), and so on. These relations would constitute a "family", although as relational objects they would have the same properties and status as all other relations of Rdegree one drawn from the PERSONS domain. Intuitively we could recognise each of these relations as actually holding information about two entity types, one of which (the LANGUAGE entity-type) had been "absorbed" into the "meta-schema" of the system with the consequence that it is no longer recognised by the query language. (We cannot use the query language to formulate the query, "list everyone who speaks French," but rather must rely solely on our "meta-schema" knowledge about the meaning of each relation.) Where the "family" of language-speaking relations has been grouped together into one relation, our necessary "meta-schema" knowledge is restricted to awareness of the existence and "intension" of only one relation.

Conversely, any "family" (in the sense described in the preceding paragraph) of M relations (M > 1) of the same relational degree N and whose attributes are drawn from set-operation compatible domains, can be "fused" into a single relation of relational degree N+1 with a new attribute having M distinct values and drawn from a (possibly new) domain whose values express the relationship holding among the original attributes. Using the example from the previous paragraph, we could "fuse" the PSL relation with anyotherrelation of Rdegree two whose attributes were drawn from the PERSONS and LANGUAGES domains, provided that their key definitions were identical, by adding a third attribute to make explicit the nature of the relationship between the entities in the first two attributes. Thus PRL and PSL could be "fused" in a new relation of Rdegree three, the new third attribute displaying values from a domain whose entities included the occurences "speaks" and "reads".

Logical schema design in practice must steer a middle course between these two possibilities, guided by the necessity not unnecessarily to multiply relations, by an anticipation of the kinds of queries users are likely to put on the data base and by an awareness of the way in which users "naturally" conceptualise the world being modelled in the data base.

2.2.5 Assumptions and Constraints

A "Base Relation" is one which cannot be derived from other relations, and has been input into the system *ab initio*. For the sake of simplicity we will impose the constraint that all Base Relations of our system exemplify one and only one relationship per relation. "Families" of relations in the above sense are also excluded. In the examples which illustrate this thesis, all relations other than Base Relations will be "derived relations," and will be considered to have been generated in response to a query. This requirement is more powerful than (but includes) the constraint that all base relations be held as reduced third normal form relations. (A reduced relation is one which cannot be split by projections into two relations which could then be joined to produce the original relation. It necessarily has a degree at most one greater than the degree of the key.) As will be demonstrated, certain queries on base relations will yield derived relations as answers to the query which do embody both types of simultaneous multiple relationships described above. Thus a tuple in a base relation will be an instance of a single relationship among its participating entities.

A further simplifying constraint, necessary if the previous one is to hold, will be the restriction of entities "eligible" for participation in the system to those entity sets such that each entity can be represented in the system by a single identifying value. (For example, a "date" would have to be represented by a Julian value, as opposed to being a concatenation of values for day, month, and year.)

With this perspective, algebraic queries on base relations can be seen as specifying operations which modify the objects (relationships and entities) whose identifiers make up the tuples of the base relations which are the arguments to the query.

2.2.6 Aims and Limitations of Reverse Translation

It is necessary to distinguish two possible sources of user confusion when dealing with syntactically and semantically valid queries which, however, may not correspond with the user's intensions.

An expression in a formal query language may be difficult to understand because it involves a large number of operations on many relations. It is reasonable to assume that, everything else being equal, increasing the number of operations making up a query will tend to increase the difficulty of understanding the meaning of the resulting relation, even if this relationship is not linear. This source of difficulty is inherent in the problem. This thesis primarily concerns itself with a second possible source of user confusion: user uncertainty about or positive misunderstanding of the semantics of a particular sequence of operations on a relation or relations. The operators of relational algebra, and their equivalents in any formal query language, are simply not in sufficiently close psychological congruence to their effects to be easily and painlessly comprehended, especially by intermittent users.

2.3 The Method of Reverse Translation

2.3.1 A Summary of the Reverse Translation Process

The Reverse Translation process accepts as input a syntactically valid query (in Relational Algebra) put to relations which have previously been created incorporating additional extra-relational information about the relationship among the entities which make up the domains taking part in the relation. To aid precision in the ensuing discussion, this "extra" information will be called the "predication" of a given relation. A "predication" as used in the remainder of this thesis will refer to the descriptive information relevant to the relationship among the entities in a relation. (The word "relationship" must be used in other contexts when discussing aspects of the Reverse Translation process, and also has an abstract flavour which is misleading when used to describe the actual strings of text which bear the information needed for Reverse Translation. Therefore we have chosen to conscript a word from philosophy which has an original meaning not opposed to the meaning we will make it use here, but which will not be confused with any other sense of the information which exists among objects.) All Base Relations are described by a single unmodified predication. Each Base Relation has a distinct predication. A derived relation will be described by one or more modified or unmodified predications. In the relation recorded in Table 2.1, for example, the predication between the "persons" and "languages" domains is represented by the strings "speaks" and "is spoken by".

The essential idea behind Reverse Translation is to add to the data base system the information which has been "factored out" of the sets of values which recorded in the system. This information -- the predication -- is held in such a way that it may be manipulated by the relational operations that are intended to be applied to the data, either before or in parallel with the execution of a query. The Reverse Translation system delivers as output one or more statements in a stylised pseudo-natural language format which may be interpreted by the user as describing the "meaning" of the relation which would be generated as a result of executing the proposed query.

It is intended that the user will be able to compare his intended abstract query (using this term in the sense elucidated in Chapter I) with the abstract query described by the Reverse Translation. Any discrepancy between the two would signal a faulty query language expression, which may then be inspected, and, if necessary, corrected.

It is <u>not</u> a goal of Reverse Translation to fool the user into thinking that he is conversing with a pseudo-human entity. The "utterances" of Reverse Translation have as their goal the conveyance of precise information about possibly complex objects. The achievement of this goal will, in certain situations, require the generation of output text which is "unnatural", in the sense that no human would speak this way "naturally". Consideration of the pragmatics of text generation for maximising human understanding is beyond the scope of this thesis, and in any case would require large-scale empirical testing in realistic conditions. It is perhaps valuable, however, to point out that the use of stylised, pseudo-natural language in situations where precision is critical and users are in the "professional end-user" category has long been acceptable: examples include real-time military and civil aviation communications.

Since the ultimate aim of Reverse Translation is to aid users to validate their queries, using computer-generated pseudo-natural language statements, there is clearly a good deal of scope for overlap with such topics as Man-Machine Interface studies, Cognitive Psychology, and Linguistics. Insights from these fields would potentially impact the <u>presentation</u> of the information carried by Reverse Translations: the particular choice of words to convey meaning, the layout of text on the screen, the possibility of using different type fonts to distinguish among objects of different types, and so on. Similarly, the modes of use of a Reverse Translation system might fruitfully be subject to many experimentally-tested design variations: some or all users might benefit from a brief training period to become accustomed to the syntax and semantics of a Reverse Translation description of a relation, for instance. But the core of this thesis is the method proposed for allowing the underlying semantic structure of derived relations to be represented in terms of the way in which their original ancestor relations were described by these relation's creators and as a proposed query might modify these descriptions. This basic method -- Reverse Translation Graphs -- lends itself to sustaining many possible implementations, with respect to the questions of text presentation and modes of use mentioned previously. The particular implementation chosen to illustrate this thesis does not claim to be based on more than the designer's intuitions, and is in any case easily modifiable.

2.3.2 Information carried by the Reverse Translation

2.3.2.1 Alternative approaches to descriptions of relations

From the standpoint of how a Reverse Translation could carry information to the user, two approaches are possible. These are identical in ultimate semantic content, but vary in their ease of interpretation.

(1) The Reverse Translation could be phrased in terms of an assertion about the derived relation _as a whole_.

(2) It could be phrased in terms of an assertion which is true for _any single tuple_ in the relation.

Where the query operation has yielded a relation consisting solely of a pre-defined entity type (such as "persons" in the example above), both approaches are equivalent as regards the ease with which a Reverse Translation can be generated and the comprehensibility of the final result from the user's viewpoint. The advantage of the second approach is seen when the final relation to be translated consists of associations of entities, such that the association considered as a single "thing" has no natural name. In such a case clarity is aided by generating a complete sentence wherein the entities involved in the relation play traditional grammatical roles.

It is worth noting that the attempts to define queries by various authorities quoted in Chapter I almost all assume that the final object to be retrieved is a set of identifiers of a single distinctly-named entity type. This is not always the case, however, and it may, incidentally, be hypothesized that the attempt to retrieve tuples which cannot be conceptualised as members of a single (composite) entity type may be particularly error-prone.

An example of the latter would be a request to retrieve tuples of degree two such that each tuple consists of the name of a person who speaks French or English, or both, and a language from that sub-set spoken by him. In practice this request will almost certainly always be given a natural language expression like "get the names of people who speak French or English, along with the languages they speak", leaving it to the user's world-knowledge to work out the particular valid query from the number of distinct valid queries which could, arguably, be a response to that request.

In this case, the relational algebra expression wanted is evidently the following:

```
A <- PSL : [ L="French"]  U PSL : [ L="English"]
```

A consists of just those tuples from PSL where the value in the "L" attribute was "French", or "English".

But any of the following expressions, consisting of additional operations on the relation A derived above, could be presented as valid solutions also.

```
(1) A1  <-  A % [P]   and   A2 <-  A % [L]
```

Persons who speak French or English, in A1, and the languages in question, in A2.

```
(2)      A3 <- A1  *  PSL
```

Each tuple of A3 consists of a person who speaks French or English, plus a language spoken by him. The latter will include French, or English, but all of his other languages as well.

```
(3)      A4 <- A3 % [ P ]   and   A5 <- A3 % [ L ]
```

The same misunderstanding as A1, but compounded with the misunderstanding of A3.

This thesis does not investigate methods of aiding users to increase the precision of their formulation of natural language expressions of abstract queries prior to submitting them to a computer for processing. Rather, it is an exploration of a method of allowing a machine to formulate a human-comprehensible expression of a relational algebra expression of a query. In order to avoid what appears to be a possible source of confusion (the attempt to speak always of relations as if each tuple were a representation of some sort of nameable entity), Reverse Translations will be complete assertions about tuples. They will be intended to be assertions which apply to any single tuple in the relation they are describing, formulated in terms of the entity types, and relationships among them, which are directly or indirectly represented in each tuple.

2.3.2.2 <u>Information about the "degree" of individual entity participation in a predication</u>

A central part of the semantics of the relationship among entities in a tuple is the question of the "degree" of the relationship. In certain queries this may be the most crucial fact, and may play a major role in causing user confusion. The term "degree" in this context must <u>not</u> be confused with the completely different concept known as the "degree" of a relation. Rather, we want to consider the "degree of an entity within a relation". The concept we are concerned with is the following: holding all other attribute values fixed, how many tuples differing only in having distinct individuals of the entity set under consideration can there legally be in a given data base? (A suggestive way of describing this concept might be to call it the "degree of freedom" of an entity in a given relation.) In a relation in third normal form, the relationships between any element of the key (assuming the key is a composite key made up of more than one attribute) is N:M; between the key as a whole and any non-key columns, N:1; between candidate keys, 1:1. In the persons-speaking-languages relation, for instance, both the person and language entity-types participate with degree N. (It would be possible to hold relationship information more precisely. For instance, to record that intra-key relationships are N:2. This possibility is not considered further in this thesis, although modifying the Reverse Translation system to incorporate such a refinement would appear to be straightforward.)

2.3.2.3 Information about selections of sub-sets of entity sets

Many queries involve the selection of sub-sets of entity identifiers which fulfill a certain condition. Where such a sub-set would replace the full range of possible entity identifiers, this fact must be incorporated into the Reverse Translation. In the previously-cited example, a query might direct the selection from the base relation of all those tuples where the language value was

"French".

2.3.2.4 <u>Information about entity set correspondence in complex associations</u>

Even a very short sequence of operations can produce a derived relation with very complex semantics as regards the relationships existing among the entities represented in it. This is especially liable to be the case where some or all of these entities are no longer represented by attributes in the final relation. Where two or more such entities from the same domain are present in a relation, there must be some way of asserting either their identity, or lack of it. An illustration of this problem, and a solution to it, are taken up in Chapter Five.

CHAPTER THREE

DATA STRUCTURES TO SUPPORT REVERSE TRANSLATION

3.1 Introduction

The goal of Reverse Translation is to be able to map any sequence of relational algebraic operations into a pseudo-natural language text which will convey to the user the meaning of the relation derived from these operations. In order to accomplish this an intermediate structure is needed which will simultaneously faithfully reflect the semantics of the sequence of operations and serve as a basis for text generation. In this chapter the intermediate data structure devised to play this role is described. First the data structure which is created by the user and which is associated with the Base Relations is shown, along with an example of the graph-creation process. Then the elaborated graphs associated with queries on Base Relations are shown for each Relational Algebraic operator, along with examples of the Reverse Translation output generated from each graph.

The information necessary to sustain Reverse Translation is represented by creating a labelled directed graph, henceforth termed a Reverse Translation (RT) graph, for each Base Relation at the time of the Base Relation's creation. With respect to its traversal by the Reverse Translation Generation algorithm, this graph is an n-ary tree made up of two kinds of nodes: Predication Nodes, and Entity Nodes. ("Non-navigational" links may also exist bewteen Entity Nodes in graphs of relations which are derived from other relations.) Relational algebra operators applied to relations may add additional Logical Operator Nodes. Predication Nodes are used to store information solicited from the relation's creator about the relationship which obtains among the entity types whose occurrences will make up the attributes of the relation. Information about these entity types is accessed via Entity Nodes, which are descendants of Predication Nodes.

The actual generation of pseudo-Natural Language Reverse Translations requires in addition to RT graphs information specific to the entities (considered in isolation from other entity sets) making up the domains of each relation. This information is input at domain creation time and held with the usual information about the domain. (The terms "entity type" and "domain" are equivalent in this thesis.) The application of relational algebra operators to relations results in the creation of several kinds of Operator Nodes. Predication Nodes, Entity nodes and Operator nodes together make up a derived graph, for the derived relation. This derived graph records the modifications that the relational algebraic operations have made in the derived relation and is used to generate the Reverse Translation.

A Base Relation Graph consists solely of a Predication

Node and Entity Nodes which are the Predication Node's only descendants. Derived graphs may in addition have one or more Operator Nodes, consisting of Logical Nodes, Comparison Nodes and Value Nodes. All information is carried in the nodes, links being used merely to navigate among them. A Reverse Translation is generated by visiting, in order, each of the nodes of the graph and generating at each node text which will in the aggregate have the form of stylised pseudo-natural language statements.

3.2 Base Relation Nodes: Entity Nodes and Predication Nodes

3.2.1 Predication Nodes

3.2.1.1 Role in Reverse Translation

Predication Nodes hold information about the kind of relationship which obtains among the values of each tuple. They express, in Sowa's terms, the "intension" of each relation. [Sowa, 1983] Each Base Relation has one and only one predication node. Predication Nodes are by far the most complex type of node, since they must bear most of the information conveyed by the reverse translation process, and because they are the node most subject to user definition.

Underpinning the Predication Node construct is the following assumption: someone who creates a Base Relation, as previously defined, must be able to generate a sentence which will transmit the meaning, or relation. (All tuples in a relation have equal status: for instance, there is no concept of ordering among tuples.) This sentence _must_ include a reference to each attribute, but a sentence which consisted _only_ of such references could not be a full description of a relation. Since attribute values are the only objects held in a relation, and since other relations with the same attributes (drawn from the same domains), in the same order, could always be created, there must be "something else" associated with each distinct relation which distinguishes it from all other relations with the same "surface structure".

This can be illustrated by the two relations representing speakers and readers of languages, PSL and PRL, whose "surface structure" of attributes is identical. The "something else" which justifies the existence of two distinct relations can be represented as strings of words which link the attribute values, and which are different for the two different relations. A PERSON _speaks_ the associated LANGUAGE in PSL, and _reads_ it in PRL. Anything which might represent the "speaks" relationship has been "factored out" of PSL precisely because it is true for every tuple. Ironically, it is precisely those strings which carry the meaning of a tuple which do not appear in a relation.

A Base Relation of Rdegree N will have a Predication Node with N Entity Nodes as descendants. The Predication Node will be termed a predication of Pdegree N. The number of immediately-descendant Entity Nodes of a Predication Node is fixed at the time of its creation by the user and does not subsequently change.

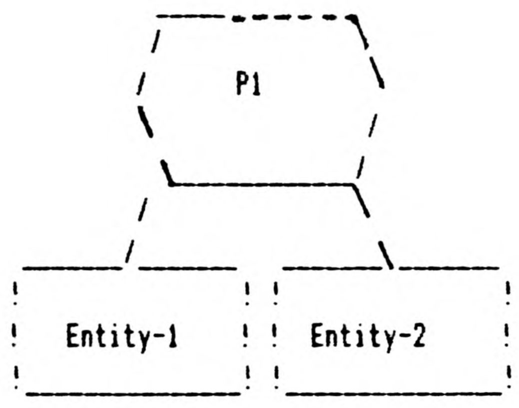


Figure 3.1. <u>A generalised Predication Node (P1) with two descendant Entity Nodes.</u>

3.2.1.2.1 Overview

A Predication Node consists of a set of Predication Phrases which are chosen by the Base Relation's creator to express the relationship existing among the values of each tuple in a Base Relation. Each phrase in the Phrase Set is in turn a set of strings. A "slot" precedes the first phrase, and follows it and all other phrases. The number of slots equals the degree of the Base Relation and the Pdegree of the Predication Node. Thus there are Pdegree-1 phrases in a Phrase Set. (Subsequent projection operations may change the degree of the relation being described by a Predication Node, but the latter's Pdegree remains constant.) This is equivalent to having one slot per Base Relation attribute. Each slot is a link to an Entity Node. Each collection of strings, when uttered with the appropriate Entity names appearing in the "slots", should be semantically equivalent to each other, differing only in that each enunciates the predication for a different permutation of the relation's attributes. (The responsibility for ensuring that this is the case belongs to the creator of the Base Relation.) The relation's creator supplies the appropriate Predication Phrase for each permutation of the relation's attributes, in both singular and plural forms, and in both positive and negative senses, at the time the relation is created. (See Figure 3.4)

It is the relation creator's responsibility to ensure that all of the sets of phrases in each sense group are equivalent to each other, and that each accurately expresses the relationship obtaining among the entity-types they link. From the system standpoint, each phrase set consists of simple, uninterpreted strings. (The system does not, for instance, hold an English grammar "knowledge base" with information about the semantics of prepositions and verbs.)

Thus a Predication Node and its descendant Entity Nodes are intended to carry either strings or references to strings such that the "meaning" of their Base Relation can be conveyed. The particular set of strings which is generated during Reverse Translation will be termed "the Reverse Translation" of a relation. Occasionally the term "the description" of a relation will be used, as a synonym. All or part of a particular Reverse Translation will sometimes be referred to as an "utterance".

Base Relations of degree N will have N-factorial semantically-equivalent predication phrase sets. Each such set consists of a <u>main predication phrase</u>, and N-2 <u>case indicator phrases</u> which link the entities not linked by the main predication phrase.

3.2.1.2.2 <u>Main Predication Phrase Set</u>

The <u>main predication phrase</u> is that phrase which carries the principal burden of communicating the "meaning" of the relation. It may be thought of as corresponding to the "verb" in a sentence. (The actual grammatical categories of the instances of domain representatives which occur in a tuple is irrelevant, because the Reverse Translation always refers to a domain representative as a member of a domain, i.e. as a noun. For example, the Reverse Translation graph of a binary relation recording VERBs in one column and possible ADVERBs modifying them in another, would have a Main Predication Phrase describing the relationship between them.)

Predication phrases will always be placed between phrases applying to entities. In a binary relation, the Main Predication Phrase will be the only component of a phrase. Note that a Main Predication Phrase may include "case indicating" prepositions, or any string whatsoever. In a binary relation which records a reflexive relationship between its two attributes, the Main Predication Phrase will be identical in both permutations. The Main Predication Phrase exists in both singular and plural forms, and in both positive and negative senses. (The conventions of English grammar do not require the other "case-indicating" phrases to be inflected for number or sense.)

Each permutation of the main predication phrase, unlike members of the Case Indicator Phrase Set described in the following section, exists in both a "positive" and "negative" sense. The "positive"sense is that which describes the relationship among the entities of a Base Relation tuple. The "negative" sense is useful in order to provide paraphrases of Reverse Translations of certain possible query expressions. For example, in a relation listing persons and the languages they read, it is necessary to be able to generate a phrase to indicate that "X reads Y", but also a phrase to communicate the negation of this fact. This can be done either by simply negating the user-supplied positive phrase ("it is not the case that X reads Y"), or by allowing the relation's creator to choose the phrase that he believes best transmits the intended negative sense (perhaps that "X does not read Y", "X cannot read Y", "X is illiterate in Y", etc.) and generating that instead. This will have advantages when confronting the problem of double negations.

3.2.1.2.3 Case Indicator Phrases Set

Case Indicator Phrases are those which, in relations of degree three and higher, are needed to supplement the information carried in the Main Predication Phrase. As with the Main Predication Phrase, these phrases may be any string whatsoever (although they typically will be, or at least include, prepositions such as "to" or "with").

Main Predication Phrase	Case Indicator Phrase
Positive Sense	
[X] supplies	[Y] to [Z]
[X] supplies	[Z] with [Y]
[Y] is supplied by	[X] to [Z]
[Y] is supplied to	[Z] by [X]
[Z] is supplied with	[Y] by [X]
[Z] is supplied by	[X] with [Y]
Negative Sense	
[X] does not supply	[Y] to [Z]
[X] does not supply	[Z] with [Y]
[Y] is not supplied by	[X] to [Z]
[Y] is not supplied to	[Z] by [X]
[Z] is not supplied with	[Y] by [X]
[Z] is not supplied by	[X] with [Y]

(X, Y, and Z are references to Entity Nodes.)

Figure 3.2 Predication Phrase Sets

3.2.1.2.4 Permutations and Mapping

The system has solicited a predication phrase set from the user at relation creation time for each permutation of the relation's attributes. Therefore, whatever permutation of the relation's attributes obtain when it is necessary to generate a Reverse Translation, the appropriate phrase set can be selected. However, a predication of degree three or higher, which has one or more arguments consisting of entities whose parent columns have been projected out, cannot simply take the permutation of its relation, as can predications of relations which have not had attributes projected out. This is because more than one permutation of the "missing" entities is possible which leaves the remaining entities in correspondence with their attributes. To see this, consider the relation SPJ, consisting of information about which Suppliers supply which Parts to which Projects. Now suppose that a relation R1 is defined, derived from SPJ by projecting out attribute P. There are three possible phrase sets which refer to the remaining entities Supplier and Project in the correct order: (Part), Supplier, Project; Supplier, (Part), Project; Supplier, Project, (Part). (The entity corresponding to the projected-out attribute is enclosed in parenthesis.) The choice of which of these phrase sets to utter is arbitrary from the point of view of their equivalence. The Reverse Translation system chooses a set which refers first to all of the remaining entities, in the order which corresponds to the order of their matching attributes, and only then to the "missing" ones. In this case there is only one such phrase set, the one corresponding to Supplier, Project, (Part). (Were the Supplier attribute to now be projected out, there would be two possible choices: Project, (Supplier), (Part) and Project, (Part), (Supplier). The choice is made by the system using an algorithm which returns the first permutation which meets the requirements stated above from the set of reverse lexicographically-ordered

Pdegree! permutations. [Page and Wilson, 1979] Reverse lexicographical ordering is chosen because in this implementation a table of the twenty-four permutations of four objects is used, rather than an algorithm generating each permutation in turn. Holding the permutations in reverse lexicographical order allows the same table to be used for permutations of fewer than four objects also, since each permutation table for N items is are "nested" within the table for N+1 items. However, this does restrict predication nodes -- and therefore Base Relations -- to a maximum of four arguments. This maximum could be expanded to five, with a table having one hundred and twenty entries. Allowing predication nodes of any number of arguments would require replacing the table-lookup approach with a (slower) permutation-generation algorithm. Since predications are not dynamic with respect to the number of arguments they take, and since a full range of queries can be illustrated with predications of three arguments, the table-lookup approach was chosen.

3.2.2 Entity Nodes

3.2.2.1 Role in Reverse Translation

Entity nodes hold information about the objects represented in relational systems by tuple values. They are the things about which something is predicated by virtue of their appearance in the relation. Each node holds a reference to a particular collection of information about the domain its entity set is drawn from. This information is not specific to any particular relation, but holds true for all occurrences of representatives of this domain in the data base. (This is described in detail in the next section.)

In addition to a domain reference, the Entity Node holds information specific to the particular appearance of occurrences of representatives of the domain in this relation: whether or not its original attribute has been projected out, or what attribute it appears under, if it has not been projected out; and its "Edegree": whether or not a particular value of a domain representative occurring in a tuple has been uniquely determined by the other tuple values, or not. Finally, when each Entity Node is originally created, it is given a unique internal identity value. (See Figure 3.4) Entity Nodes of derived relations may have descendant nodes carrying information about them.

3.2.2.2 Terminology

An Entity Node whose corresponding attribute has not been projected out will be termed a Participating Entity Node. An Entity Node whose corresponding attribute has been projected out will be termed a Non-participating Entity

Node.

A SELECTION operation may add information about an Entity Node. The new information added as the result of a SELECTION will be held in a graph, which will be termed a Qualification Sub-graph. The Qualification Sub-graph is a descendant of that Entity Node whose corresponding attribute has been the argument of a selection expression. This is described in detail in Sections 4.4.3 and 5.2-5.3.

3.2.2.3 Predication Roles

If the Entity Node is going to play a role in the relation such that it could more usefully be Reverse Translated with a name other than its domain name, then this alternative name is held in the Entity Node. This "Predication Role name" can be helpful in overcoming ambiguity where a relation has more than one attribute drawn from the same domain. In a Base Relation the requirement that each attribute have a distinct name prevents intra-entity ambiguity from occurring, especially if the relation's creator chooses attribute names which are suggestive of the role played by the entities filling that attribute. But if one or more of the common-domain attributes gets projected out, the attribute name as such is no longer available. In this case the Predication Role name helps maintain the distinction between entities from the same domain.

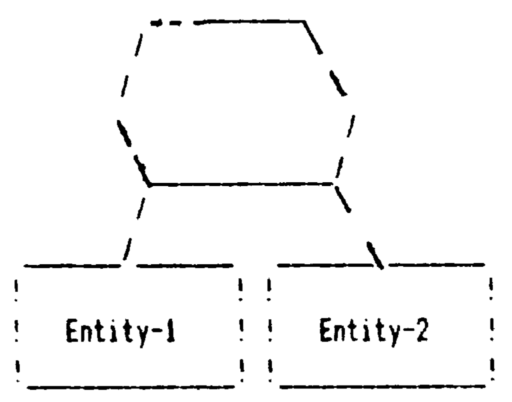


Figure 3.3. Entity nodes.

3.2.2.2 Domain information

Each Entity Node refers to a particular domain. Each domain in the system have the following information recorded about it at domain creation time:

entity set name, both singular and plural forms.

animation flag, is an individual of this domain "animate"? The only practical effect of this is in the choice of pronouns: animate domains are referred to by "who", inanimate ones by "which".

representation flag: whether or not individuals are represented in this domain by their "own" names, or by special codes. If they are represented, the singular and plural forms of their representative's names are recorded. As an example, languages are recorded directly using their own names ("French", "German", etc.) but suppliers are represented by Supplier Codes. This information is used when uttering qualifying graphs: thus the system will describe "a Language, which is French," but "a Supplier, whose Supplier Code is S123".

<u>self-identification flag</u>: a Boolean value which is TRUE if the values which represent individual entities can always appear in a Reverse Translation without a prior reference to their domain or domain-representative. If they can be, the domain is said to be "self-represented" and the Reverse Translation generation algorithm may optionally omit reference to the domain (or domain representative) from which the values are taken. An example of the distinction in practice would be, assuming that the domain of persons is not self-representing but the domain of languages is, a Reverse Translation fragment consisting of "at least one PERSON, whose <u>name</u> is Smith, speaks [a LANGUAGE, which is] French." In this fragment, domains are in upper case, domain representatives are underscored, and the part of the Reverse Translation phrase which can be optionally omitted (given that the domain of LANGUAGES is self-representing) is enclosed in square brackets.

<u>data type and range data</u>: what datatype the values which will represent individual entities in the system will be.

comparison phrases: a default set of phrases for the relational comparison operators ("<", "=", etc.) is available, but the user can override these and supply his own if a tailor-made set of phrases would add clarity. (This is most likely to occur where a domain's values are judged to be self-identifying, as described above. For instance, objects and values from a domain of AGE might be usefully compared with the phrases " is younger than", "is", etc.)

3.2.2.3 Domain Information Storage

All information about domains is stored in an array of domain records where it may be accessed by the Reverse Translation generator whenever the latter must utter information relating to a particular entity node.

3.3 An Example of a Base Relation Graph and its Reverse Translation

3.3.1 Generalised RT Base Relation Graph

```
          ____________________________________
         / 1: <predication> (a phrase set)     \
        /  2: <permutation> (1..pdegree!)       \
       /   3: <sense>   (positive or negative)   \
       \   4: <pdegree> (= number of descendant  /
        \                  nodes -- fixed)       /
         \______________________________________/
          /                !
         /                 !
 _______/_______   ________!________   ________ ________
! 1: <domain>   ! !                 ! !                 !
! 2: <Edegree>  ! !                 ! !                 !
! 3: <attribute>! !                 ! !                 !
! 4: <id>       ! !                 ! !                 !
! 5: <role>     ! !                 ! !                 !
 ---------------   -----------------   -----------------
```

Figure 3.4 Information Held in Predication and Entity Nodes

3.3.2 An Instance of a Base Relation Graph

SPJ

```
                       ______________________________________
                      / 1: <predication>: (See Figure 3.2)   \
                     /  2: <permutation>: 1                   \
                    /   3: <sense>       : positive            \
                    \   4: <pdegree>     : 3                   /
                     \                                        /
                      \______________________________________/
                      /                 !
                     /                  !
             _______/______      _______!_______     ______  ______
<domain>     ! SUPPLIER    ! ! PART           ! ! PROJECT        !
<Edegree>    ! N           ! ! N              ! ! N              !
<attribute>  ! 1 [SNO]     ! ! 2 [PNO]        ! ! 3 [JNO]        !
<id>         ! 003         ! ! 023            ! ! 019            !
<role>       ! none        ! ! none           ! ! none           !
             !             ! !                ! !                !
             !_____________! !________________! !________________!
```

Figure 3.5 The Base Relation Graph for SPJ

3.3.3 Generating a Reverse Translation from an RT Graph

3.3.3.1 Discussion and Justification of the Method

A Reverse Translation is a generated by an in-order traversal of a Reverse Translation Graph. At each node, text is output which depends on the state of the node, and for Derived Relation Graphs (discussed in the following chapters) on the particular state of a "sense marker" which is global to the node. The principles involved may be seen by an examination of the considerations involved in choosing the wording of each phrase generated from the graph illustrated in Figure 3.5.

3.3.3.2 Predication Phrases

The Predication Phrases ("supplies", "to" etc.) are simply those supplied by the relation's creator, with the proper sense, number and permutation.

3.3.3.3 Indicator Phrases

The user must be able to see to which attribute value an Entity Node whose attribute has not been projected out is referring, and must be able to distinguish between these Entity Nodes and those whose attributes have been lost through projection. If a relation has attributes from common domains, generating their domain names alone will not distinguish between them. Both of these objectives can be accomplished simultaneously by including in the Reverse Translation of Participating Entity Nodes, the name of the attribute to which they correspond. This is done by enclosing the attribute name in square brackets and putting it immediately after the Entity Node's domain name in the Reverse Translation. As a possibly useful measure of linguistic redundancy, the domain name of Participating Entity Nodes is also preceded by the Indicator Phrase, "the indicated". This is done to reduce the possibility of one-character attribute names in the Reverse Translation being overlooked.

3.3.3.4 Edegree Phrases

The degree of participation of the entity in the predication is signalled by an "Edegree phrase" which is the final phrase in the Reverse Translation of the Entity Node. For Participating Entity Nodes of Edegree N the phrase is "and possibly other" followed by the domain name.

3.3.4 Reverse Translation of the RT Graph of SPJ

```
The indicated Supplier [SNO], and possibly other Suppliers,
supplies
the indicated Part [PNO], and possibly other Parts
to
the indicated Project [JNO], and possibly other Projects
```

3.4 Derived Relation Nodes: Logical Nodes, Comparison Nodes, and Value Nodes

3.4.1 Logical Nodes

The Logical Nodes are those which result from one of the three set operations, UNION, INTERSECTION, and, DIFFERENCE, and from the JOIN. UNION creates an OR Logical Node, DIFFERENCE a NOT Logical Node, while both INTERSECTION and JOIN create an AND Logical Node. A Logical Node is not intended to represent the algebraic operation which created it, but rather holds a token for the logical relationship brought into being between the RT graphs of the operand relations by the relational algebraic operation which operated on their parent relations. This token is illustrated in Figure 3.4 by "<L>". A Logical Node becomes the root node of the (derived) RT graph which is composed as the result of the operation. A Logical Node always has as descendant graphs in such a case the graphs of the two operand relations. The left descendant graph is the graph of the relation on the left of the operator in the Relational Algebra expression, and the right-hand graph is the graph of relation to the right of the operator. The complete graph of the derived relation may sometimes be transformed into an equivalent graph, possibly with fewer constituent elements, according to rules explained in Chapter Five. As a result of the algorithms which create Reverse Translation graphs, OR nodes can also be the parent nodes of Comparison Nodes, or of OR or AND nodes, while AND nodes can be the parent nodes only of Comparison Nodes or

AND nodes. Examples of these graphs used to represent derived relations can be found in the following chapter, with a justification for the special status of OR nodes.

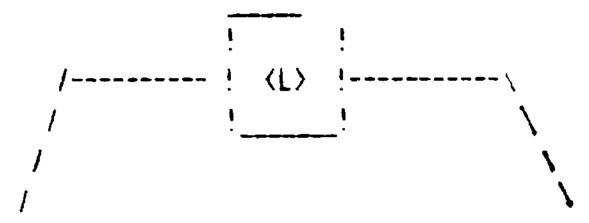


Figure 3.4. A Logical Operation Node.

3.4.2 Comparison nodes

A Comparison Node is a node with a single descendant Value Node, holding a token for the comparison operator used in a SELECTION operation, illustrated by "<c>" in Figure 3.4. In the SELECTION expression,

```
      R1  <-  P1  : [P="French"]
```

a Comparison Node holding a token for the "=" operator is created.

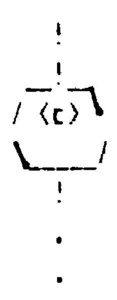


Figure 3.4.1. Comparison Node.

3.4.3 Value nodes

A Value Node is a simple leaf node which holds a string equivalent of the value in a SELECTION operation (illustrated in Figure 3.4.1 by "<v>" . In the SELECTION expression,

R1 <- P1 : [P="French"]

a Value Node holding the string "French" is created.

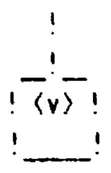


Figure 3.6 <u>A Value Node</u>.

CHAPTER FOUR

RELATIONAL OPERATIONS AND REVERSE TRANSLATION

4.1 Introduction

This chapter introduces the actual process of Reverse Translation by examining the effect of each relational operation on an example Base Relation Graph (BRG), (defined in Sections 3.1 - 3.2) illustrating the "before" and "after" states of the graphs associated with the parent relation(s) and the derived relation.

A relation which is derived from a monadic relational algebra operation (a PROJECTION, PERMUTATION, or SELECTION on a single relation) will inherit a graph which is composed of a modified variant of the ancestor relation's graph. (See Figures 4.1 - 4.2) A relation derived from a dyadic operation (UNION, DIFFERENCE, INTERSECTION or JOIN) will inherit a graph composed from both graphs of its parent relations. (See Figures 4.3 - 4.9)

The derived graph of the resultant relation is emphatically not an "execution tree", although occasionally it may bear a superficial resemblance to one. (An execution tree would add a new node for each new operation, with the last operation being the root node of the tree. An RT graph, although a tree, and affected by each new operation in which its parent relation takes part, does not necessarily have a new node added at the root of the tree as the result of an algebraic operation, and in fact may not have new nodes added at all.)

For the derived relation's graph to be of value to the user, it must serve as the input to a Reverse Translation algorithm. A given RT graph can be Reverse Translated in more than one semantically-equivalent way, given the conventions of English. It may also be subject to prior manipulations which preserve its semantics but enhance the user's comprehension of the graph's meaning when its Reverse Translation is generated. The full Reverse Translation system incorporates both further graph manipulations and additional linquistic choices incorporated in the Reverse Translation algorithm itself, which are not illustrated in this chapter. The pragmatic considerations that guide the choices made in further graph manipulation and in the generation of a Reverse Translation are taken up in detail in Chapters Five and Six. In this chapter each derived graph is accompanied by the Reverse Translation which would be generated in the absence of further meaning-preserving manipulations and without regard to possible simplifications in the generation of its Reverse Translation.

4.2 PROJECTION

4.2.1 Effect on relation semantics

A PROJECTION strips out one or more attributes from a relation. In terms of the user-supplied tuple-description of the relation, a PROJECTION withdraws the specific occurrence of an entity value from each "slot" in the relation corresponding to the attribute or attributes projected out. If before a PROJECTION a relation told us that John speaks French and English, and Mike English and German, then projecting out the attribute holding the language values leaves us with a relation from which we can learn only that John and Mike speak one or more languages.

PROJECTION is the operation which, viewed from the Reverse Translation perspective, has the most radical impact on relation semantics. This will be seen to be especially so, where the attributes projected out have been created by compounding two or more attributes from distinct relations in previous intersections, differences or joins. This will be demonstrated in Chapter Five.

4.2.3 Effect on Graph

The effect of a PROJECTION on the RT graph of the derived relation is to set the relevant attribute value of the Entity Node of each discarded entity-set in each Predication Node to zero. (To enhance the visual impact of the PROJECTION operation, the convention by which the effect of PROJECTION on an Entity Node will be illustrated in this thesis is a small open box at the point of attachment between the projected-out Entity Node and the link by which it is attached to its Predication Node. This is merely a presentation device.) The effect of PROJECTION is illustrated by Figure 4.1 where the Base Relation SPJ is projected on attributes SNO, PNO, with attribute JNO (representing Projects) being discarded. All information relating to JNO is lost from the tuples of the resulting relation, SP, but is retained in the graph of SP . In the case of a PROJECTION on a Derived Relation Graph (DRG) to produce a further DRG, where a given projected-out attribute may be referred to by more than one Entity Node (the descendants of different Predication Nodes), every such Entity Node's attribute value must be set to indicate a missing attribute.

4.2.4 Example Queries

```
SP <- SPJ % [ SNO, PNO ]      (Graphs illustrated in
                                    Figure 4.1.)

P <- PC % [ P ]               (Graphs not illustrated.)
```

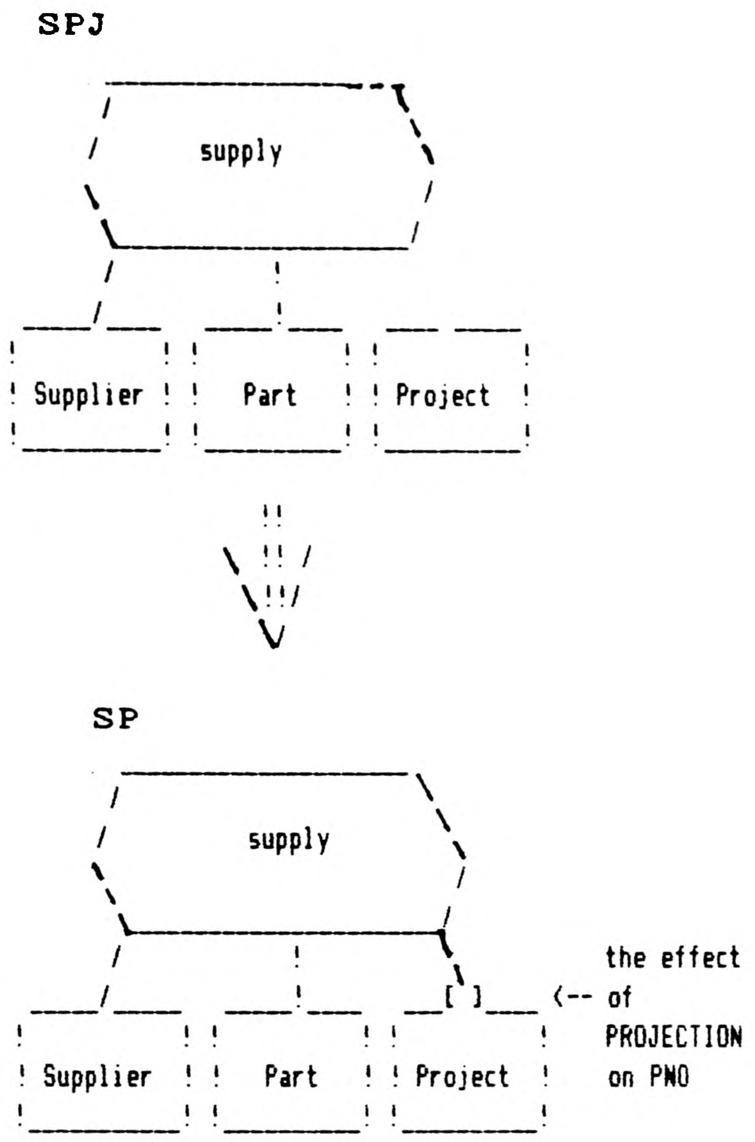


Figure 4.1 The effect of a PROJECT:

4.2.5 Effect on Translation

The disappearance of an entity through PROJECTION may impact its Reverse Translation in three ways.

(1) An entity which has been projected out of a tuple (a Non-participating Entity Node) must now have its current absence (but previous existence) signalled by a different Indicator Phrase. The exact phrase used to replace it depends on several factors which are discussed in Chapter Seven. The Indicator Phrases shown here are those which are generated for Base Relation Graphs. The two possibilities, corresponding to Entity Nodes of Edegree

1, and Nodes of Edegree N, are discussed in Section 4.2.5.1.

(2) The Attribute Reference (" ([<attribute name>]) ") must vanish, since the corresponding attribute has been eliminated through the PROJECTION.

(3) The Edegree Phrase may be changed, depending on the Edegree of the affected Entity Node (1 or N). Both cases are discussed, and the choices made justified, in the next two sections.

4.2.5.1 <u>The Edegree of Entities</u>

4.2.5.1.1 <u>Entities of Edegree N</u>

In the case of a BRG, the Indicator Phrase for a PROJECTED-out Entity Node of Edegree N will be "at least one". This particular Indicator Phrase incorporates the Edegree information which, for Participating Entity Nodes, is carried in the Edegree Phrase ("and possibly others"), which can, consequently, be dropped in the interests of minimising verboseness. This is illustrated in Section 4.2.6.1.

4.2.5.1.2 <u>Entities of Edegree 1</u>

An Entity Node of Edegree 1 in a BRG cannot have an Indicator Phrase of "at least one" since it represented in fact the <u>only</u> entity occurrence which could have appeared in this column. Its Indicator Phrase will accordingly be "a/an". Unlike the case of Entity Nodes of Edegree N, the Edegree information is not clearly signalled by the new Indicator Phrase, and so the Edegree Phrase is retained. This is illustrated in the example in Section 4.2.6.2.

4.2.6 Reverse Translation Examples

4.2.6.1 Reverse Translation of SP

The indicated Supplier [SNO], and possibly other Suppliers,
supplies
the indicated Part [PNO], and possibly other Parts,
to
<u>at least one</u> Project

4.2.6.2 Reverse Translation of P

The indicated Person [P], and possibly other Persons,
was born in
<u>a</u> Country, <u>and it alone</u>

4.3 PERMUTATION

4.3.1 Effect on relation semantics

The PERMUTATION of a relation A to produce a relation B will affect B's semantics with respect to its participation in any operation whose outcome is dependent on the order of the participant relation's attributes. In the syntax of the relational algebra used to illustrate this thesis, this is the case for the set operations UNION, DIFFERENCE and INTERSECTION. For instance, consider a relation of degree two, with both attributes drawn from the domain of PERSONS, representing the relationship of teacher-student between the first and second attributes. Now consider a second relation, also of degree two and with both attributes drawn from the domain of persons, representing parents and offspring. If the first relation is intersected with the second, the resultant relation represents all pairs of PERSONs such that the first person teaches and is the parent of the second. If the first relation is now permuted (so that they "change places"), then its intersection with the second (parents-offspring) relation gives a relation representing all pairs of PERSONs such that the first person teaches and is the offspring of the second.

4.3.2 Effect on Graph

The PERMUTATION of a relation's attributes will have two effects on its graph:

(1) it will notionally re-set the order of the Predication Node's pointers to the entity argument slots. The practical effect of this will be to present the descendant Entity Nodes of the Predication Node to any processing algorithm in a revised order, reflecting the permutation of the relation. (Figure 4.2)

(2) it will change the permutation indicator value and therefore will alter the particular predication

Phrase Set which will be generated in the Reverse Translation. As described in Section 3.2, the Predication Node has been supplied by the user, at relation creation time, with a set of descriptive phrases for each permutation of the entity arguments of the relation. The current phrase set is indicated by the permutation indicator value.

4.3.3 <u>Example Query</u>

JPS <- SPJ # [JNO, PNO, SNO]

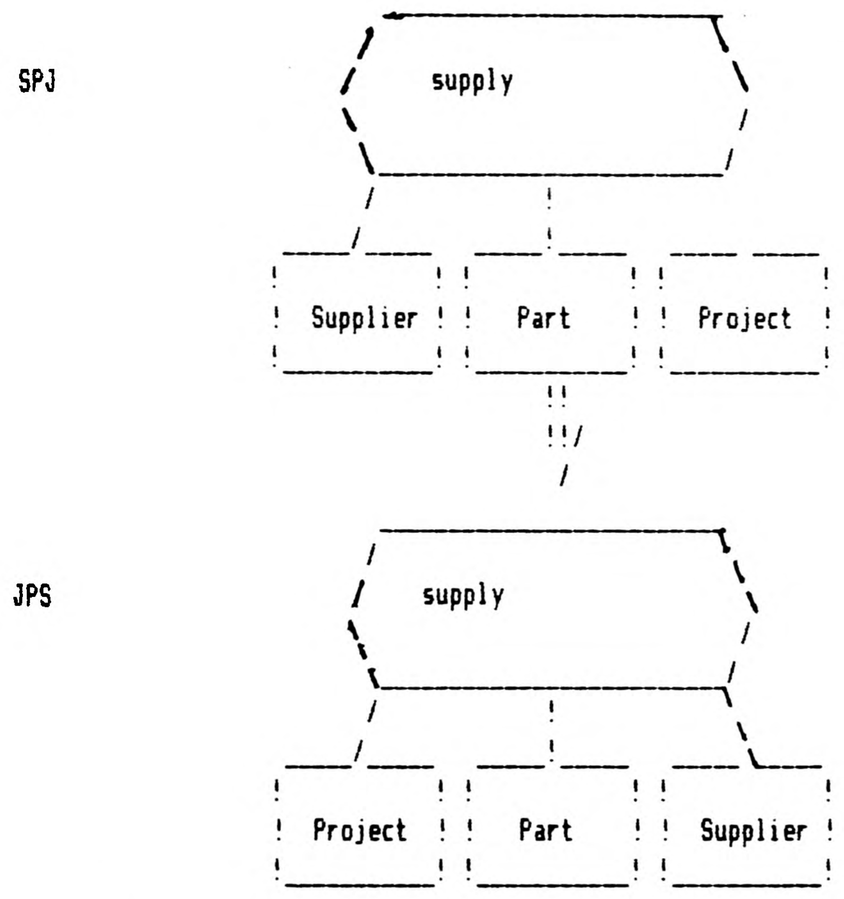


Figure 4.2 <u>The effect of a PERMUTATION.</u>

The order of entity appearance is altered to correspond to the new attribute order, and the appropriate predication phrase (corresponding to the new entity order) will be generated.

4.3.5 Reverse Translation of JPS

```
The indicated Project [JNO], and possibly other Projects,
is supplied with
the indicated Part [PNO], and possibly other Parts
by
the indicated Supplier [SNO], and possibly other Suppliers
```

4.4 SELECTION

4.4.1 Effect on relation semantics

A SELECTION chooses a sub-set of the tuples of the operand relation. In terms of the user-supplied description of the tuples of a relation B created as the result of SELECTION on a relation A, a SELECTION provides us with additional information about the entity occurrence represented by the attribute selected on. In a Base Relation we know only that what is true for the domain from which entity representative occurrences are drawn is true for each entity occurrence. In the derived relation we are able to assert additionally that the value of each occurrence of an entity representative corresponds in a manner specified by the SELECTION expression to a sub-set of the possible entity representative values which might occur in this position.

Thus the practical effect of SELECTION on a domain with well-ordered values is to tell us the following: _either_ what particular domain value every occurrence of that domain in this attribute _is_, for a SELECTION expression with the equality operator; _or_, to tell us either or both of the inclusive or exclusive upper and lower _limits_ of the domain values which may be taken by occurrences found in this attribute (the lesser-than, lesser-than-or-equal, greater-than-or-equal and greater-than operators), with zero or more values which the occurrence may not have (the not-equality operator).

There is more than one way to represent such knowledge of permitted and non-permitted domain values internally. The method chosen in the work reported in this thesis was to hold the information in a way maximally consistent with the rest of the RT apparatus. This is as a graph.

4.4.3 _Effect on RT Graph_

A SELECTION on a given attribute in a Base Relation Graph has the effect of attaching a Qualification Sub-graph to the selected-on attribute's Corresponding Entity Node. A SELECTION made on an attribute which has not previously been the argument in a SELECTION expression adds a single Comparison Node with a Value Node as its sole descendant to the Corresponding Entity Node in the

4.4.4 Example

```
SPJ1  <-  SPJ  : [ PNO > "P1" ]
```

4.4.5 Effect on translation

The effect of attaching a Qualification Sub-graph to an Entity Node is to insert a qualifying phrase (underscored in the example below) into the Reverse Translation following the entity reference and preceding the Edegree phrase. If the domain which is represented by the thus-modified Entity Node is one whose entity occurrence values have been declared at domain-creation time to be represented by values from another domain, then the phrase "whose" is generated, followed by the of the name of the domain's representative, followed by the RT phrase for the value of the Comparison Node, followed by the string held in the Value Node.

The cases of more complex Qualification Sub-graphs, and of Qualification Sub-graphs after their parent Entity Nodes have been made Non-Participant nodes through projections, are taken up in Chapter Five.

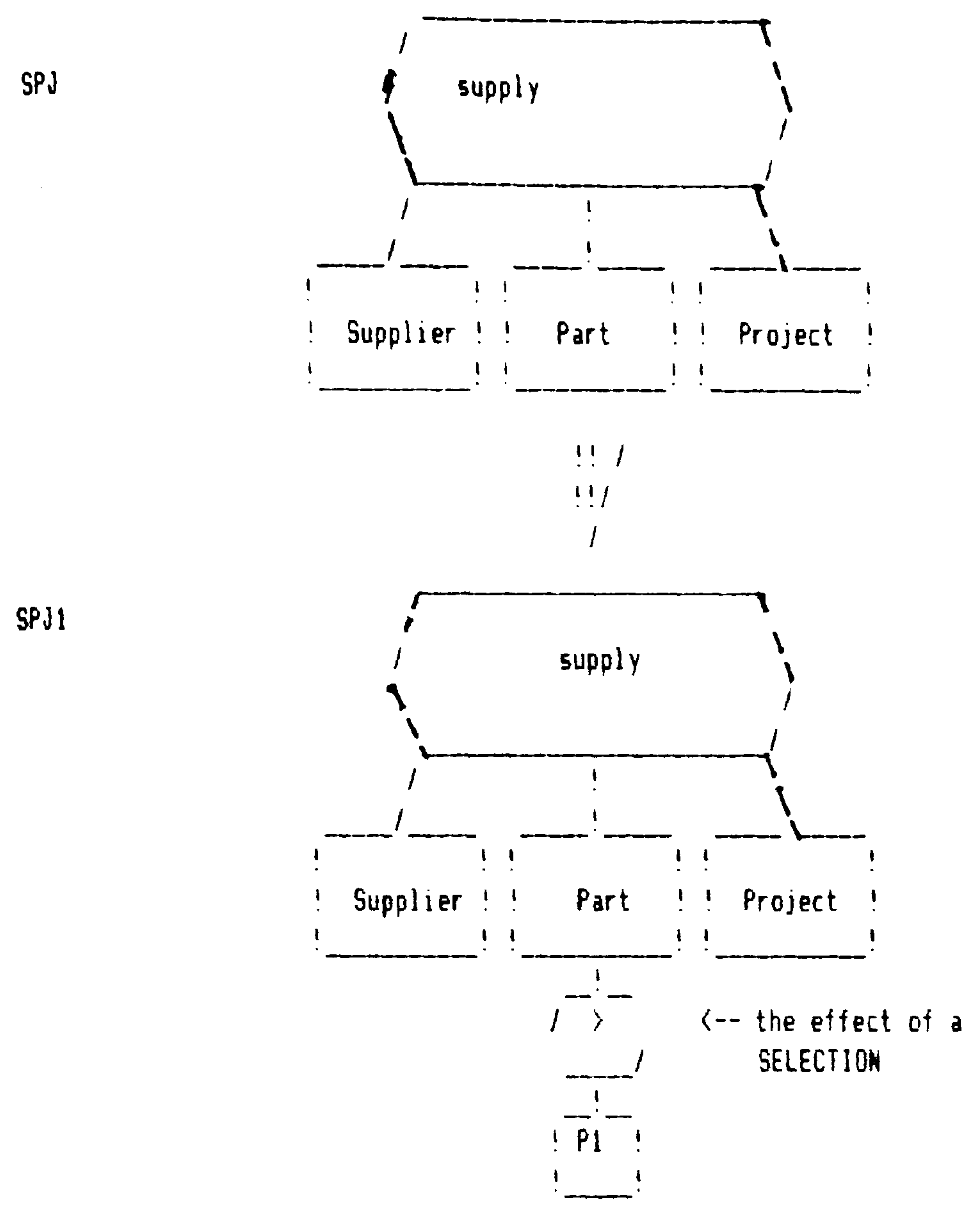


Figure 4.3 <u>The effect of a SELECTION on an unqualified Entity Node.</u>

4.4.6 <u>Reverse Translation of SPJ1</u>

> The indicated Supplier [SNO], and possibly other Suppliers,
> supplies
> the indicated Part [PNO],
> <u>whose Part-number is greater than P1,</u>
> and possibly other Parts
> to
> the indicated Project [JNO], and possibly other Projects

4.5 <u>The Set Operations</u>

The set operations will be illustrated two relations: One will be the relation PSL, PERSONs and the LANGUAGEs they <u>speak</u>, introduced in Chapter Two and illustrated in Table 2.1. The other will be the relation PRL,

4.5.1 INTERSECTION

4.5.1.1 Effect on relation semantics

The effect of intersecting two relations A and B to produce a relation C is to produce a relation in which the following can be said of each of its tuples: everything which was true for a tuple of relation A is true for a tuple of relation C, and everything which was true for a tuple of relation B is true for a tuple of relation C.

4.5.1.2 Effect on RT Graph

The effect of intersecting two Base Relation RT graphs is to unite them via an AND Logical node, and to link each participating Entity Node (any Entity Node which corresponds to an existing attribute in its relation) in the left-hand graph to its equivalent in the right-hand graph. This is illustrated in the diagrams by arcs between node boxes, but actually implemented via setting the unique node identity value of each participating Entity Node in the right-hand graph to the value of its equivalent in the left-hand graph. The necessity for this linkage, and its effect on the Reverse Translation of the derived graph is explained in Section 4.5.1.7.

The resulting list of (notionally) linked Entity Nodes will be termed an <u>Entity Node Chain</u>, and the left-most Entity Node will be termed the <u>PrincipalEntityNode</u>. In the immediately-following example graph there are two Entity Node Chains. The Entity Nodes referring to the same (non-projected) attribute will be termed <u>Corresponding Entity Nodes</u>. The Entity Nodes in a given Entity Node Chain will be termed <u>Equivalent Entity Nodes</u>. Equivalent Entity Nodes are all Corresponding Entity Nodes (assuming that they refer to a non-projected out attribute) but not all Corresponding Entity Nodes are necessarily Equivalent Entity Nodes. Note that Equivalent and Corresponding Entity Nodes must be drawn from the same domain (although not all Entity Nodes drawn from the same domain are necessarily Equivalent or Corresponding Entity Nodes).

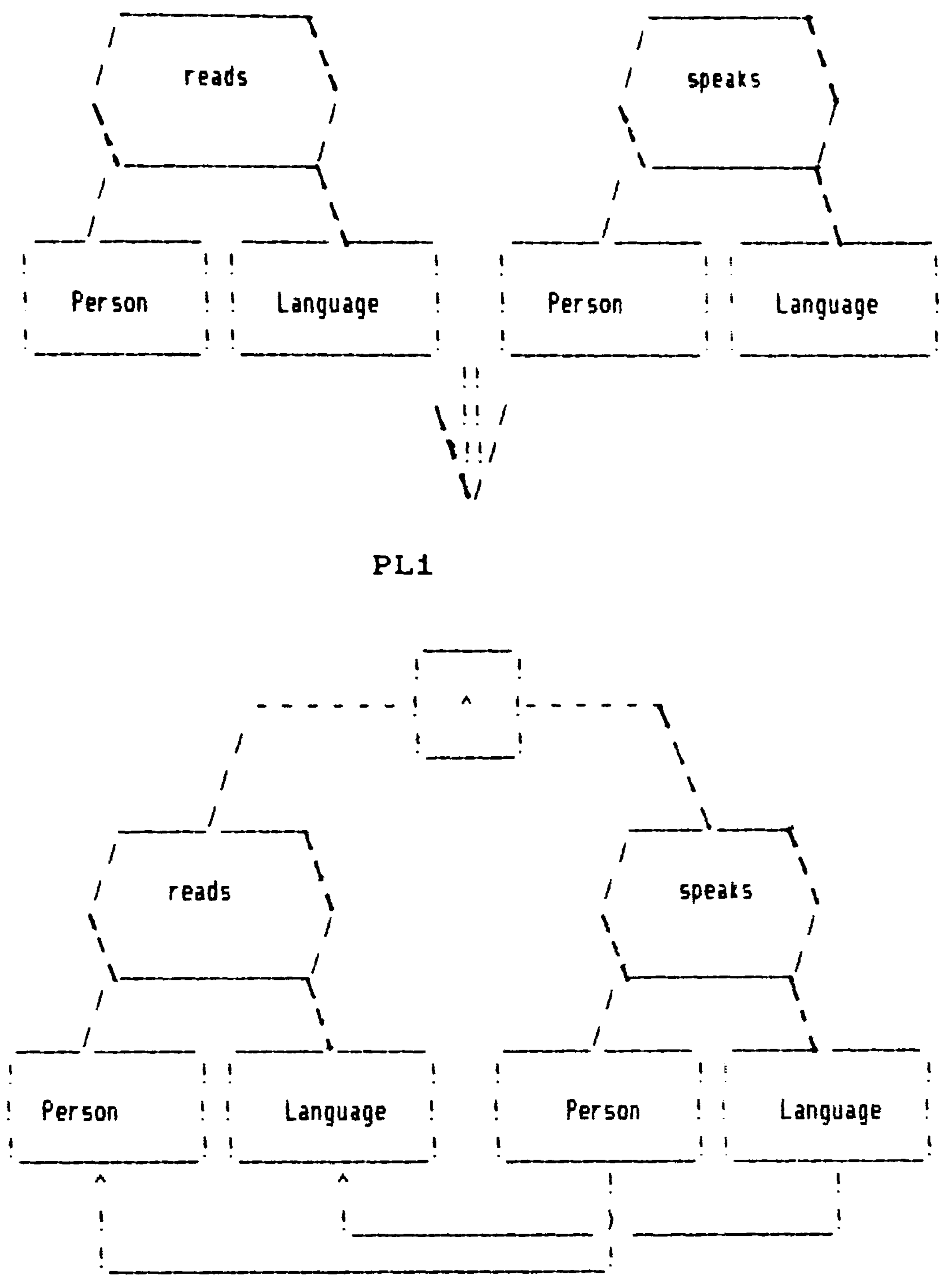
Although not illustrated in this chapter, a further process of transferring Equivalent Entity Nodes' Qualification Sub-graphs, if such exist, to the Principal Entity Node will also take place, as described in Chapter Five. The information which is thereby recorded, linking Equivalent Entity Nodes, will be used by the RT generation algorithm, as explained in Chapter Six.

4.5.1.4 Example Query

PL1 <- PRL ^ PSL

4.5.1.5 Effect on translation

Where the derived graph has not been reduced, the left descendant graph will be uttered. Then a linking phrase is generated, which will be "and" in the case of RT Graphs derived strictly from Base Relation Graphs. Then the right descendant graph will be uttered. The existence of an Entity Node Chain (Entity Nodes in the right-hand graph corresponding to an Entity Node in the left hand graph) has no effect on the Reverse Translation where the Entity Nodes concerned represent attributes which have not been projected out. The justification for this is the following: where a Reverse Translation is about an existing entity representative, it is sufficient to make reference to the particular attribute it occurs in, to be unambiguous about which possible entity representative is being referred to. The opposite case is considered in the next Section.



(Note that Entity node links are not implemented as pointers but rather via equating Entity Node identity values.)

Figure 4.4 <u>**The effect of INTERSECTION**</u>

4.5.1.6 Reverse Translation of PL1

The indicated Person [P], and possibly other Persons,
reads
the indicated Language [L], and possibly other Languages,
<u>and</u>
the indicated Person [P], and possibly other Persons,
speaks
the indicated Language [L], and possibly other Languages

4.5.1.7 Entity Node Chains and Relation Semantics

The crucial role of Entity Node Chains in maintaining a true representation of the semantics of relational operations may be seen if we consider the following two sequences of operations:

Sequence I

```
P1 <- PSL % [P]
P2 <- PRL % [P]
R1 <- P1 ^ P2
```

Sequence II

R2 <- PL1 % [P] (See Section 4.5.1.4 and Figure 4.4)

R1

P [PERSONS]
Adam
Gunther
Uli
Zahid

R2

P [PERSONS]
Adam
Gunther
Uli

Table 4.2. The relations R1 and R2.

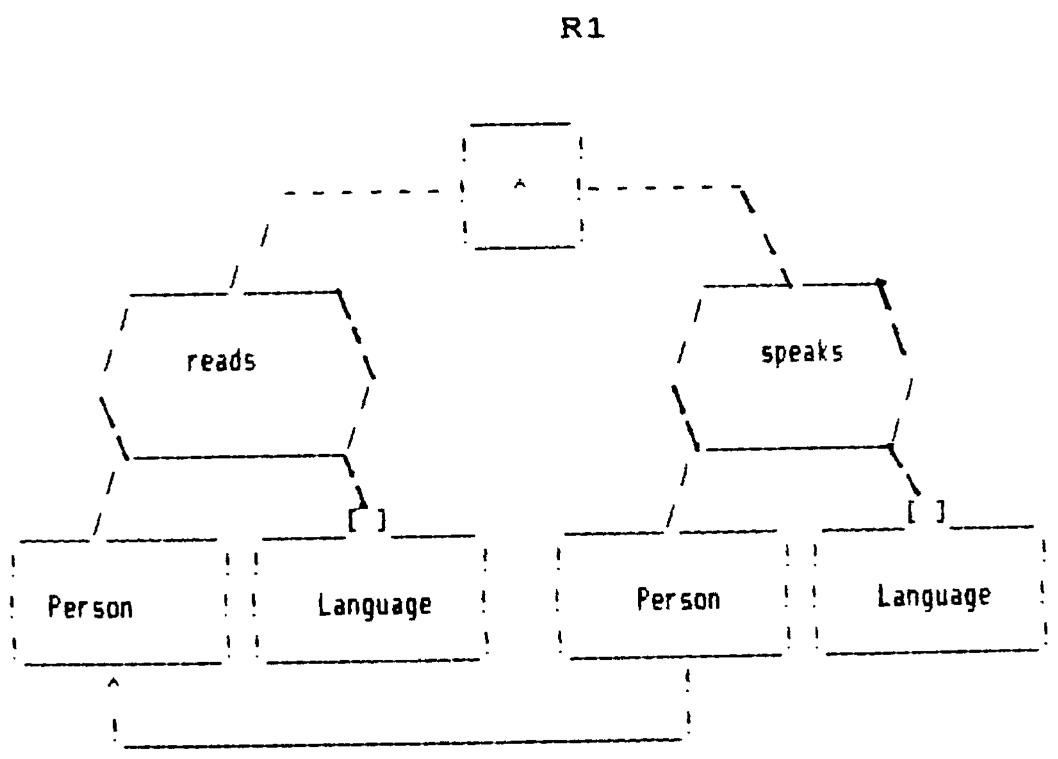


Figure 4.5 The RT Graph of R1

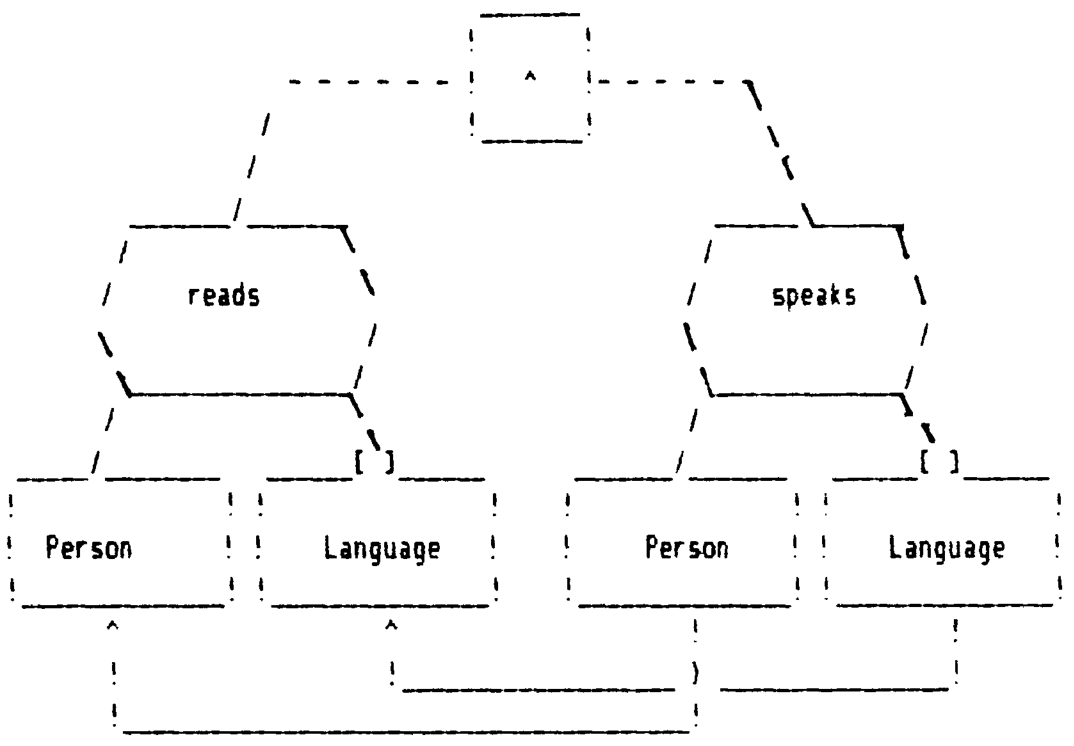


Figure 4.6 The RT graph of R2

The differences in these two graphs represent crucial differences in the semantics of the sequence of relational operations: PROJECTIONs, on the one hand, and DIFFERENCE, INTERSECTION, and JOIN, on the other, are not commutative. The two relations R1 and R2 are distinguished by the fact that the first was created by projecting and then intersecting, and the second was created by intersecting and then projecting. If both sequences of operations yielded identical RT graphs, then these graphs would not be accurate representations of the semantics of their relations. The practical effect of switching the order in which PROJECTION is done is to make R1 contain all those people who speak one or more languages, and read one or more languages (but not necessarily the same ones), while R2 contains those people who speak one or more languages <u>which</u> <u>they</u>

also read. This critical distinction, resulting from a single reversal in the sequence of relational operations, would appear to be fertile ground for user confusion. The Reverse Translation process must be able to handle the distinction.

The problem can be seen to arise in the text generated for any Non-principal, Non-participating Entity Node. Recalling the Reverse Translation for the graph of the relation PLi (Figure 4.4), we see that projecting out the Language Entity Nodes (yielding R2) will raise the question: how to indicate that the Language referred to in the second predication is the same Language as the one referred to in the first predication, now that we are deprived of the ability to refer to its attribute name?

R1 does not present such a problem. The Languages read and spoken are not necessarily the same ones, and thus the second mention of a Language requires no reference back to the previous mention of a Language. (Of course, the Indicator Phrase given must not imply such a connection.)

Reverse Translation of R1

> The indicated Person [P], and possibly other Persons,
> reads
> at least one Language,
> and
> the indicated Person [P], and possibly other Persons,
> speaks
> at least one Language

In the case of R2, such a reference must be included

Node is uttered, it must be put on a stack of previously-uttered Entity Nodes. Prior to uttering an Entity Node, we must search this stack to see if a Node with the same Node-id (that is, one which precedes the current node in a chain) has already been uttered. If so, the Indicator Phrase must refer to this previously-uttered node. The effect of this solution is seen in the Reverse Translation of R2.

Reverse Translation of R2

> The indicated Person [P], and possibly other Persons,
> reads
> at least one Language,
> and
> the indicated Person [P], and possibly other Persons,
> speaks
> that Language

4.6 DIFFERENCE

4.6.1 Effect on relation semantics

Theeffect of differencing relation B from relation A to produce a relation C is to produce a relation for which the following can be said for each of its tuples: everything which was true for a tuple of A is still true for a tuple of C, but whatever was true for a tuple of B is not true for a tuple of C.

4.6.2 Effect on RT Graph

descendant the graph of the relation being differenced from (the relational expression which occurs on the left side of the NOT Operator), and as its right descendant the graph of the relation being differenced. Participating Entity Nodes in the right-hand graph are linked in an Entity Node Chain to their equivalent Entity Nodes in the left-hand graph, in the fashion illustrated in Section 4.6.

4.6.3 Terminology

A predication which is the immediate right descendant of a NOT Logical Operator will be termed a Negated Predication. (This is not to be confused with the "negative sense" set of Main Predication Phrases held in the Predication Node, although the negative sense phrases are intended to provide an optional means of generating the Main Predication Phrase of a Negated Predication.)

4.6.4 Example Query

```
PLd <- PRL - PSL
```

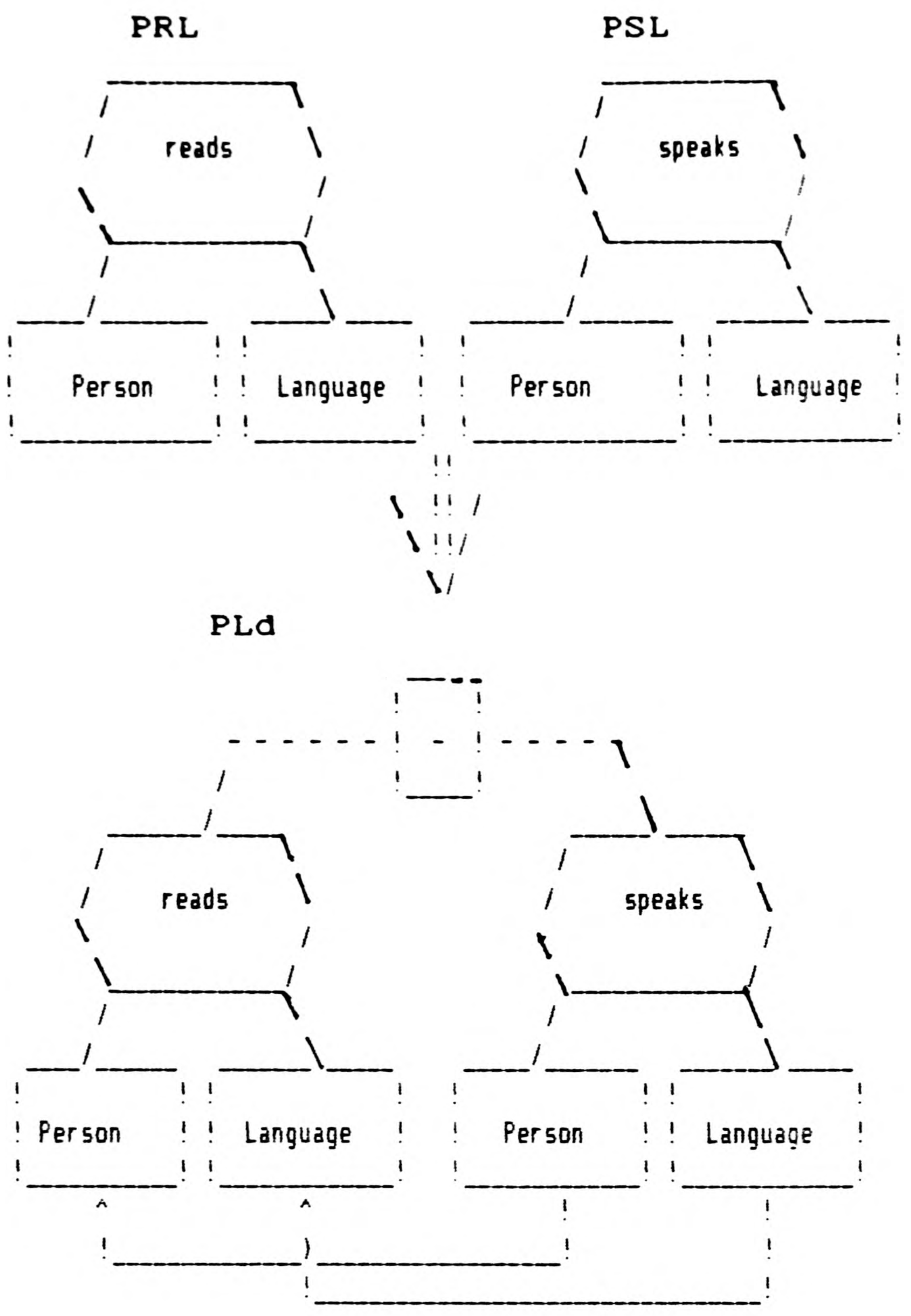


Figure 4.7. <u>The effect of a DIFFERENCE.</u>

4.6.5 <u>Effect on translation</u>

If no transformations are carried out on the deri graph, then its Reverse Translation will simply be Reverse Translation of the left descendant of the node, followed by the conjunction phrase "but it is the case that", followed by the utterance of the ri descendant Predication Node. The latter is a Nega Predication. Its descendant Entity Nodes will have th Reverse Translation effected in the following ways:

(1) The Indicator Phrase of Non-participating Entity Nodes of Negated Predications is altered from "at least one" to "any". Were no alterations to be made, we would be presented with Reverse Translations like "but it is not the case that the indicated Person ... speaks <u>at least one</u> Language", which does not convey the required meaning so clearly as the word "any".

(2) The Edegree information of each Participating Entity Node is omitted. It is superfluous, if it is the same as the Edegree information of the left-hand graph's Equivalent Entity Nodes, and incorrect, if it is different. Were the key of PSL to be the attribute <u>P</u> alone, then the Reverse Translation of the LANGUAGE Entity Node would be "the indicated Language [L], and it alone." But this would not be correct if carried over to the Entity Nodes of the Negated Predication in the derived relation PLd, as it would possibly lead the user to infer that the negation applied to the Edegree phrase, rather than to the whole predication. The Reverse Translation fragment "but it is not the case that the indicated Person [P], and he alone, speaks the indicated Language [L], and it alone," raises the possibility that multiple speakers and/or multiple-language speaking capacity is being asserted, which is not the case. Therefore the Edegree phrases of Non-principal Entity Nodes which are the immediate descendants of negated

predications are omitted.

4.5.2.6 Reverse Translation of PLd

> The indicated Person [P], and possibly other Persons,
> reads
> the indicated Language [L], and possibly other Languages,
> <u>but it is not the case that</u>
> the indicated Person [P],
> speaks
> the indicated Language [L]

4.5.3 UNION

4.5.3.1 Effect on Relation Semantics

The effect of unioning a relation A with a relation B to produce a relation C is to produce a relation for which the following can be said for each of its tuples: either everything which was true for a tuple of A is still true for a tuple of C, or everything which was true for tuple of B is still true for a tuple of C. "Or" is "weak" or "inclusive" or, and thus both assertions may be valid for a tuple of C.

4.5.3.2 Effect on Graph

A UNION operation on Base Relation graphs creates a graph which is composed of the graphs of the parent relations as left and right descendants from an OR Logical Node. [Fig. 4.8] Corresponding Entity Nodes (Entity Nodes referring to the same attribute, as defined in Section 4.5.1.3) in the left and right descendant graphs are <u>not</u> linked by an Entity Node Chain. (Note that this is in contrast to their treatment in INTERSECTION and DIFFERENCE operations.) This is because the predications in the left-hand graph do not necessarily apply to the entities represented in the right-hand graph, and _vice versa_. For any given tuple (the object of all Reverse Translation text), all that can be said is that either the Reverse Translation of one operand relation describes it, or the Reverse Translation of the other does, or possibly both do.

4.5.3.3 Example Query

PLu <- PRL U PSL

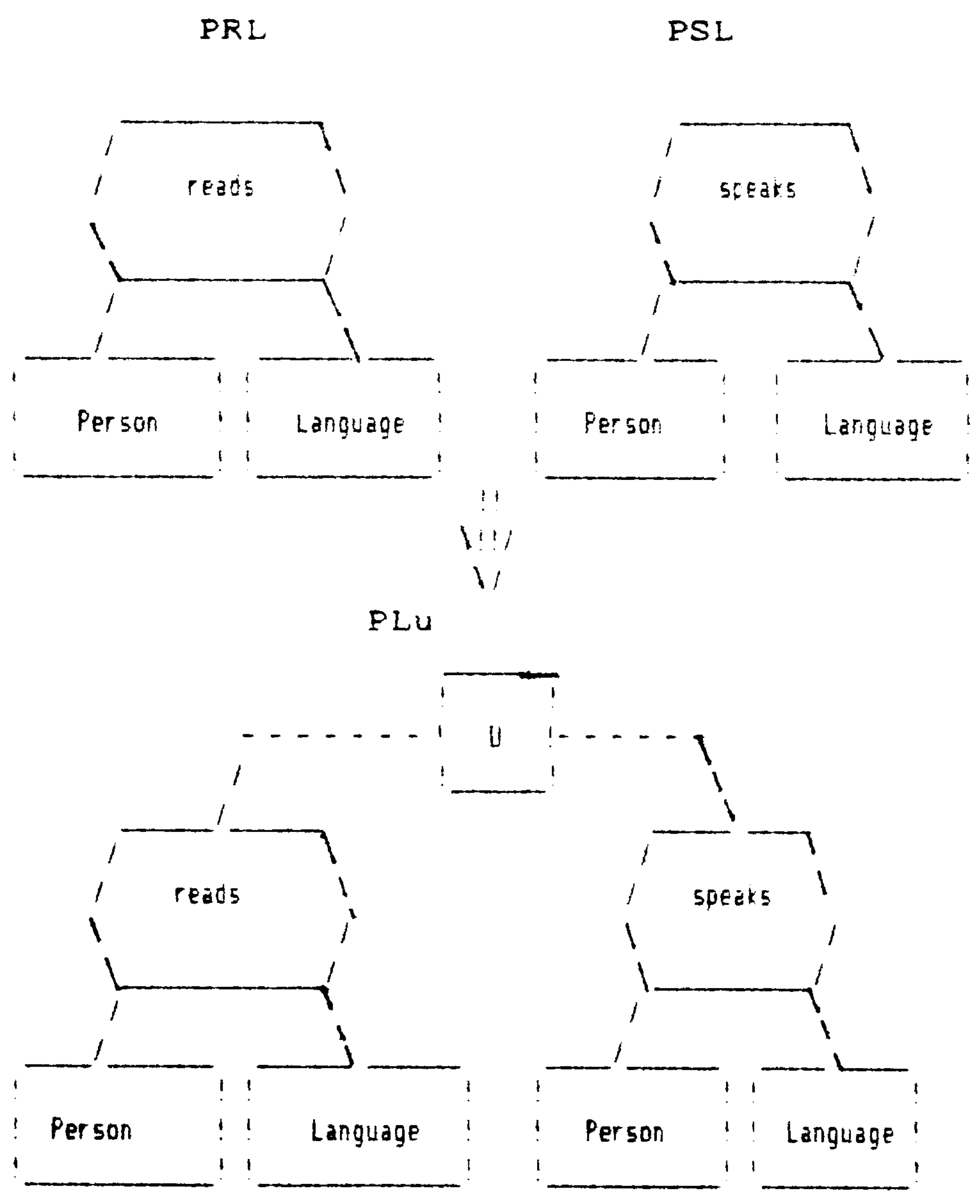


Figure 4.8. The effect of a union.

4.5.3.4 Effect on translation

The phrase "either" is uttered, followed by the utterance of the left descendant graph, followed by the utterance of the phrase "or", followed by the utterance of the right descendant graph, followed by the phrase "or both".

4.5.3.5 Reverse Translation of PLu

> either
> the indicated Person [P], and possibly other Persons,
> reads
> the indicated Language [L], and possibly other Languages,
> or,
> the indicated Person [P], and possibly other Persons,
> speaks
> the indicated Language [L], and possibly other Languages
> or both.

4.6 JOIN

4.6.1 Effect on Relation Semantics

The effect of joining a relation A with a relation B over a common attribute to produce a relation C is to produce a relation for which the following can be said for each of its tuples: everything which was true for a tuple of A is still true for the inherited partial tuple of A which forms part of a tuple of C, and whatever was true for a tuple of B is also true for its incarnation as part of a tuple of C. The two part-tuples making up a complete tuple of C share a common tuple value.

4.6.2 Effect on RT Graph

A JOIN operation over two relations unites the graphs of the participating relations with an AND Logical Node, and connects the Equivalent Entity Nodes, which correspond to those attributes in each relation over which the JOIN has been made, in an Entity Node Chain. This is done with notional identity links by setting the right-hand graph's Entity Node identity values equal to the left-hand graph's corresponding Entity Node identity values. Note that an intersection operation is identical to a JOIN where all attributes participate in the JOIN. It is instructive to compare the graph illustrated in Figure 4.9 with, in the first instance, that of a full INTERSECTION (illustrated in Figure 4.4), and in the second instance, that of a zero-common-attributes JOIN, (illustrated in Figure 4.10). The only visible distinction among the respective RT graphs is in the notional links between Equivalent Entity Nodes in the left- and right-hand graphs, but the semantic differences (which RT must communicate) are crucial.

4.6.3 Example Query

(To conform to the required syntax for the JOIN operation, we must assure that attributes not participating in the JOIN have different names. This is done using the rename facility, which is illustrated first.)

```
PSL @ [ L -> LS ]   (Rename the non-participating
PRL @ [ L -> LR ]    attributes.)
PSRL <- PSL * PRL
(JOIN  PSL and PRL over the commonly-named attribute, P.)
```

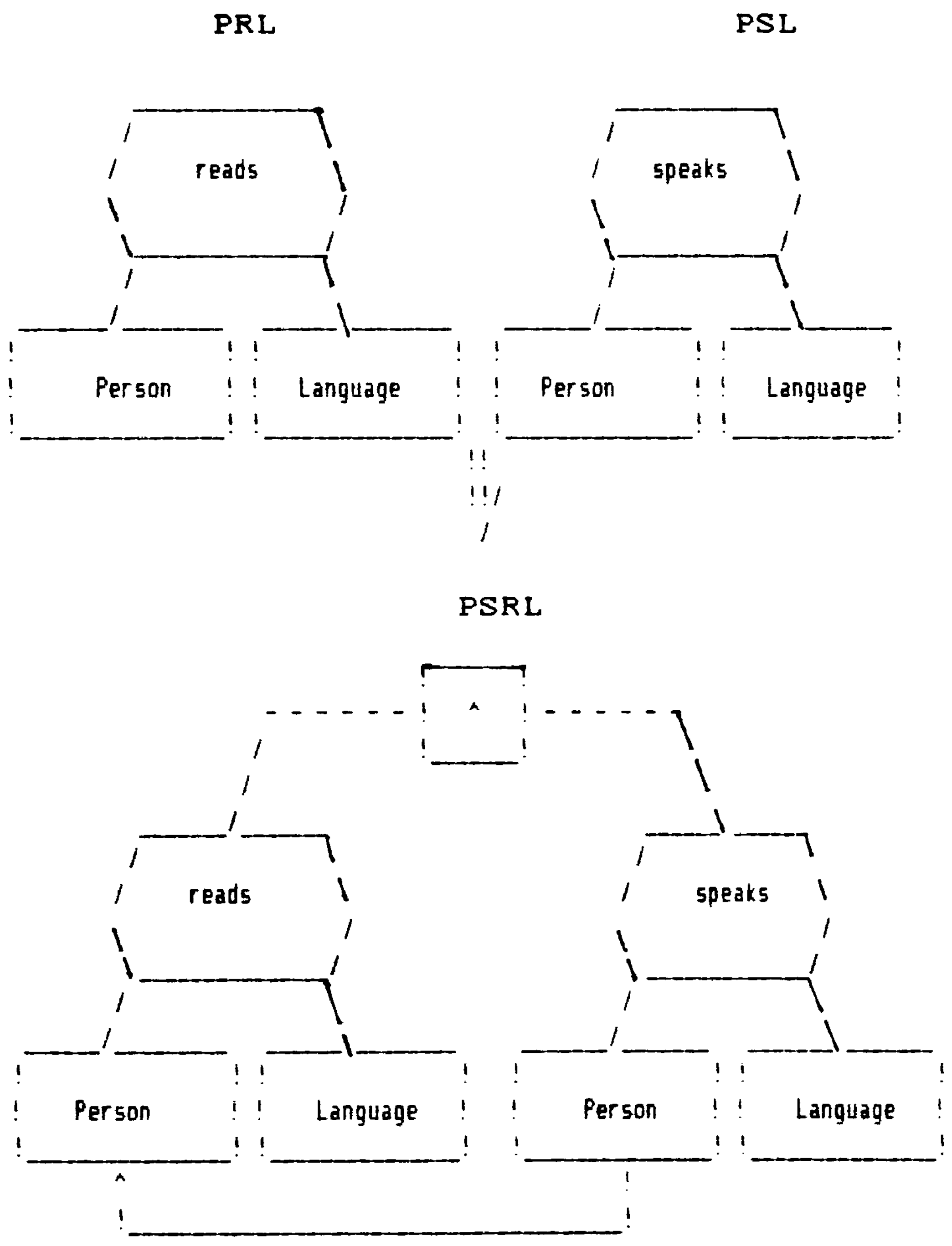


Figure 4.9 RT Graph of the relation shown in Table 2.8

4.8.4 Effect on translation

Because the root node of an RT graph produced by a JOIN operation is an AND Logical Node, the algorithm for generating its Reverse Translation is identical to that for INTERSECTION operations: the Reverse Translation of the left-hand descendant graph will be generated, followed by "and", followed by the Reverse Translation of the right-hand graph. Note that in a JOIN, unlike an intersection, not all attributes are merged, and thus not all Participating Entity Nodes are linked. In this example this is reflected in the Reverse Translation's generation of the attribute names following each entity. The generation of the attribute names is crucial for drawing the user's attention to the distinction between the first, spoken, Language, (in attribute LR), and the second, read, Language, (in attribute LS). It is instructive to compare the Reverse Translation for the above operation to the Reverse Translation for the intersection of the same two relations. In the latter case, "the indicated language" is the <u>same</u> language, the single value appearing in the column of the attribute <u>L</u>. In the operation under scrutiny here, there are two distinct languages being referred to (whether or not they have the same value): the language held by attribute <u>LR</u>, and the language held by attribute <u>LS</u>.

The question then arises, what happens if the attributes <u>LS</u> and <u>LR</u> are projected out, leaving only attribute <u>P</u>? The solution to this problem is taken up in the next Chapter.

4.8.5 <u>Reverse Translation of PSRL</u>

The indicated Person [P], and possibly other Persons,
speaks
the indicated Language [LS], and possibly other Languages,
<u>and</u>,
the indicated Person [P], and possibly other Persons,
reads
the indicated Language [LR], and possibly other Languages.

4.8.6 <u>Cartesian Product</u>

4.8.6.1 <u>Effect on RT Graph and Translation</u>

A JOIN of two relations with no commonly-named attributes results in a Cartesian product of the two relations. The RT graph of the derived relation is exactly identical to the RT graph of the equi-join illustrated above, but there are no "sideways links" between Entity Nodes, because there are no shared attributes. Entity Node links denote that the Entity Node in question is described by more than one Predication.

This graph may be compared to the RT graph of a UNION. The graph of a UNION is likewise devoid of Entity Node links, because it asserts that a tuple of the derived relation is described by one <u>or</u> the other of the descriptions of its parent relations. A Cartesian JOIN asserts that a tuple of the derived relation is described by <u>both</u> descriptions, but that the first description applies to the first elements of the new tuple (derived from the left-hand parent relation), the other description applying to the remainder of the tuple (derived from the other parent).

```
PSL  @ [ L -> LS ] (Rename all attributes. As there are
PSL  @ [ P -> PS ]  now no commonly-named attributes a
                    full Cartesian product will result
PRL  @ [ L -> LR ]  when PSL and PRL are JOINed.)
PRL  @ [ P -> PR ]

PSPRL  <-  PSL  *  PRL
```

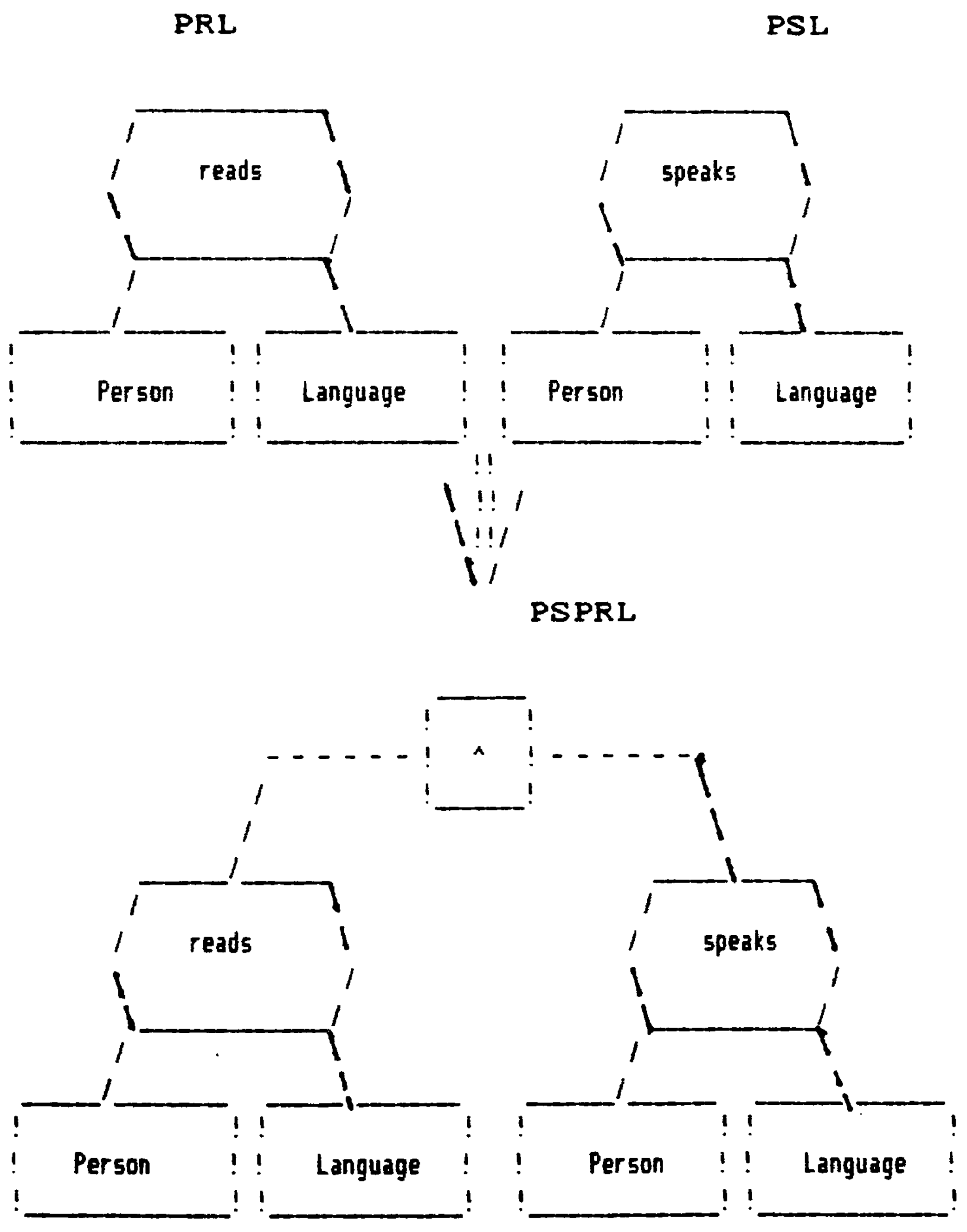


Figure 4.10 <u>The RT Graph of the relation shown in Table 2.9</u>

4.8.6.2 Reverse Translation of PSPRL

The indicated Person [PS], and possibly other Persons,
speaks
the indicated Language [LS], and possibly other Languages,
<u>and</u>,
the indicated Person [PR], and possibly other Persons,
reads
the indicated Language [LR], and possibly other Languages.

CHAPTER FIVE

ENTITY QUALIFICATION

5.1 Introduction

Chapter Four introduced the concept of Qualification Sub-graphs modifying Entity Nodes as a method of representing the effect of a SELECTION. In that chapter, only the attaching of a Qualification Sub-graph to an Entity Node in a Base Relation Graph was considered. An Entity Node in a Base Relation Graph does not have a Qualification Sub-graph prior to participating in an operation, and thus the attaching to it of a descendant graph is a straightforward matter. This chapter completes the discussion of Qualification Sub-graphs by extending it to cover all cases where an Entity Node might be modified by the attachment to it ofa Qualification Sub-graph. The ground prepared in this chapter will be a necessary foundation for discussing the processing of derived relations in the general case, which is taken up in the following chapter.

5.2 Terminology introduced in this Chapter

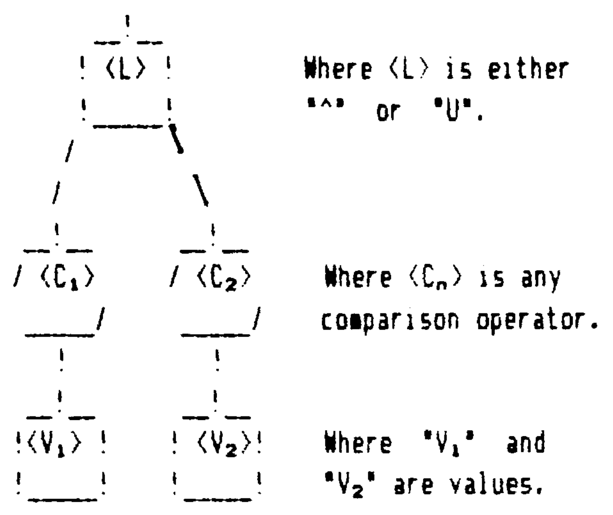
5.2.1 Qualification Sub-graph types

A Qualification Sub-graph [introduced in Section 3.2.2.2] will be termed a Simple Comparison Sub-graph if its root node (pointed to by the Entity Node which it modifies) is a Comparison Node. A single Logical Operator Node with two Simple Comparison Sub-Graphs as descendants will be called a Simple Logical Sub-graph. All other cases, where more than one Logical Node (AND or OR nodes) appears in the sub-graph, and which will not have a Comparison Node as the root node, will be termed Complex Qualification Sub-graphs.

5.2.2 Canonical form

In the course of creating RT graphs, whenever two Simple Comparison Sub-graphs are first united by a Logical Node, the resulting pair of (leaf) Value Nodes are put into "canonical form". This is done by comparing the left-hand Value Node to its right-hand Value Node, and reversing pointers from the Logical Node if the left-hand Value Node holds a value which is lexicographically greater than the right-hand Value Node. Note that the concept of lexicographical ordering operates at the level of the internal representation of the entity identifiers, and thus is valid even for those domains which are declared by the user not to be ordered. (For example, the value "French" is less than the value "German", even if the domain of languages has been declared to be one which does not permit such comparisons at the query language level.)

Holding Sub-graphs in canonical form is necessary for the efficient operation of Sub-graph reduction, described in the next section.



<L>
Where <L> is either "^" or "U".
<C₁>
<C₂>
Where <Cₙ> is any comparison operator.
<V₁>
<V₂>
Where "V₁" and "V₂" are values.

This graph is in canonical form iff $V_2 >= V_1$.

Figure 5.1 A canonical form for Simple Logical Sub-graphs

5.2.3 Sub-graph Reduction

If a Simple Comparison Sub-graphs in Canonical Form is replaced by a single Simple Logical Sub-graph with the same truth value, then the former Sub-graph will be said to have been reduced.

5.2.4 Sub-graph Transferral

If a Qualification Sub-graph which is a descendant of an Entity Node is detached from that node and attached to another Entity Node, the graph is said to have been transferred.

5.3 Attaching Qualification Sub-Graphs to each other

5.3.1 Introduction

The example of a SELECTION in Chapter Four was on a Entity Node which had not yet been the object of a

SELECTION, or of any other operation which would attach a Qualification Sub-graph to it. Attaching a Qualification Sub-graph to a previously "un-Qualified" Entity Node thus did not involve any graph manipulation. In this section the problem of attaching Qualification Sub-graphs to Entity Nodes which already have Qualification Sub-graphs is addressed. As the next chapter will demonstrate, the necessity for attaching a Qualification Sub-graph to an already modified Entity Node arises when a SELECTION occurs on that Entity Node's corresponding attribute, or when a JOIN or INTERSECTION operation adds qualified Entity Nodes to an Entity Node Chain through Sub-graph Transferral.

First the case of Simple Comparison Sub-graphs is illustrated, followed by the case of Complex Qualification Sub-graphs. Then the notion of Sub-graph reduction is explained.

5.3.2 Attaching Simple Comparison Sub-Graphs to Simple Comparison Sub-graphs

Attaching a Simple Comparison Sub-graph to an Entity Node having a descendant Simple Comparison Sub-graph will create a Simple Logical Sub-graph, with each of the original Sub-graphs descendants of an AND Logical Node. (This graph may, however, be reduced as explained in Section 5.4.) The justification for this is straightforward. The original Qualification Sub-graph carried information about the particular values that the modified Entity Node's representatives in a tuple can take. The information carried by the new Qualification Sub-graph must be added in a way that preserves the previous information. An uncomplicated way to do this is simply to unite the two graphs by making them descendants of an AND Logical Node. This allows us to take advantage of the fact that we already have created the apparatus to create and process trees which descend from AND Logical Nodes, including the generation of Reverse Translations from them. (See Chapter 4.5.1.)

As an example, consider the relation SPJ1 (Section 4.4.2), which consists of all those tuples from SPJ such that the Part Numbers representing Parts in the PNO attribute are greater than 1. Let SPJ2 be the relation consisting of all tuples from SPJ1 such that the Part Numbers representing Parts in the PNO attribute are less than 6.

```
SPJ2 <- SPJ1 : [ PNO < 6 ]
```

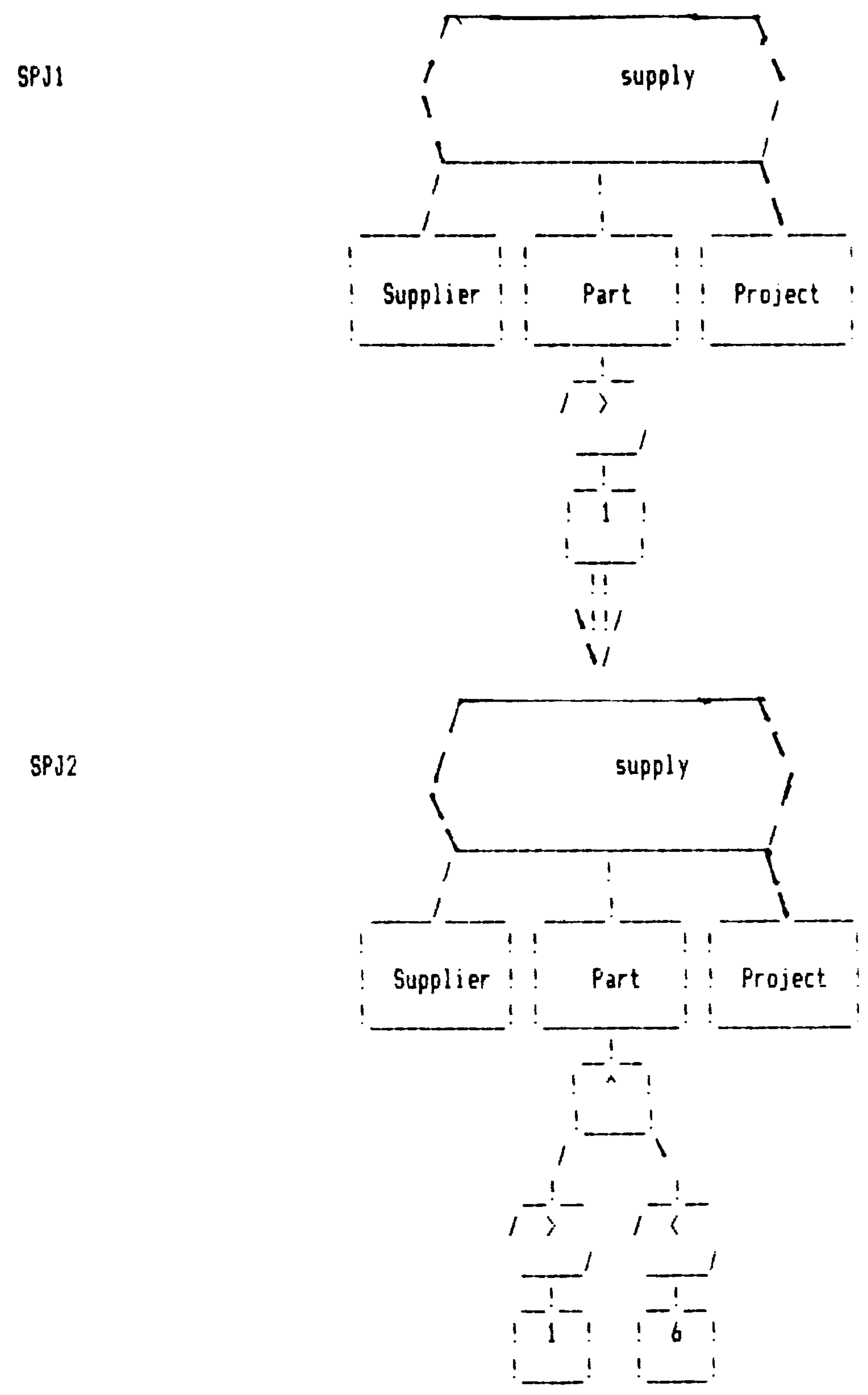


Figure 5.2 Attaching a Simple Comparison Sub-graph to an Entity Node with an existing Simple Comparison Sub-graph

5.3.3 Reverse Translation of SPJ2

```
The indicated Supplier [SNO], and possibly other Suppliers,
supplies
the indicated Part [PNO],
whose Part-number
is greater than 1
and
is less than 6
and possibly other Parts
to
the indicated Project [JNO], and possibly other Projects
```

5.4 Reduction and Checking of Qualification Sub-Graphs

The previous example assumed that the AND-composition of two Simple Comparison Sub-graphs was a straightforward operation, as indeed it was for the comparison operators and values (" > 1" and " < 6") selected. However, we must consider other cases. There is nothing to prevent the user from constructing query expressions which would yield the AND-composition of Simple Comparison Sub-graphs with such comparisons as (" > 1" and " > 4"). This particular example would in fact be equivalent to a single Simple Comparison Sub-graph with the comparison "> 4". There are two query paths by which this position could be reached.

Sequence I

```
S1 <- SPJ : [ SNO > 1 ]
S2 <- S1  : [ SNO > 4 ]
```

Sequence II

```
R1 <- SPJ : [ SNO > 1 ]
R2 <- SPJ : [ SNO > 4 ]
R3 <- R1 ^ R2
```

Either of these would yield the following Simple

Logical Sub-graph attached to the Principal Supplier Entity Node:

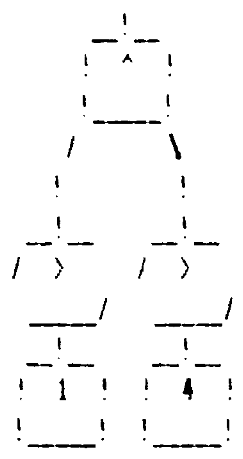


Figure 5.3 Redundant Information

The fragment of Reverse Translation that would be generated therefrom, is

```
the indicated Supplier [SNO],
whose Supplier-Code
is greater than 1
and
is greater than 4
```

This is undesirable for two reasons. Firstly, it is redundant, since if a Supplier's Supplier-Code is greater than 4, it is necessarily also greater than 1 and it is unnecessary to say so. Secondly, and more serious, a casual reading by a user might lead him to believe that the derived relation contained the Supplier Codes of Suppliers whose Supplier Codes were greater than 1, and the Supplier Codes of Suppliers whose Supplier Codes were greater than 4. The decision to make a Reverse Translation take the form of an assertion about a single tuple, and thus make the text generated from Qualification Graphs refer explicitly to single individuals, was motivated in part to overcome problems of this kind. (This is a classic illustration of the difference between the way in which the terms "and" and "or" may be understood in the casual use of natural language, as contrasted to their precise use in logic.)

It may be noted here that Sequence I could be a perfectly valid sequence of operations, motivated, perhaps, by a desire to first view the subset of all tuples with Suppliers whose Supplier Number was greater than 1, and then to view the subset within this subset of all those with a Supplier Number greater than four. Getting this sequence backwards (as in Sequence IA) yields a final relation S2A with the same semantics as S2, but should signal user confusion, since the final operation cannot change its operand relation S1A.

Sequence IA

```
S1A <-  SPJ  : [ SNO > 4 ]
S2A <-  S1A  : [ SNO > 1 ]
```

But an even worse mistake can occur, which can be seen if we consider the following sequences:

Sequence III

```
S1 <- SPJ : [ SNO > 4 ]
S2 <- S1  : [ SNO < 3 ]
```

Sequence IV

```
R1 <- SPJ : [ SNO > 4 ]
R2 <- SPJ : [ SNO < 3 ]
R3 <- R1 ^ R2
```

In Sequence III, S2 will necessarily be an empty relation, since S1 can have no Suppliers whose Supplier

Codes are less than 3. (Like Sequence IA, the final operation could not affect its operand relation.) Likewise, in the semantically-identical Sequence IV, R1 and R2 are disjoint, and therefore R3 will necessarily be an empty relation. Either of these queries could result from user confusion about the effect of the "AND" (INTERSECTION) operator in relational algebra, which is by no means a formal counterpart of the word "and" in English. It may be counter-intuitive that to get a relation containing both Suppliers whose Supplier Codes are greater than 4, <u>and</u> Suppliers whose Supplier Codes are less than 3, we must <u>not</u> use the AND (INTERSECTION) operator.

Within a "pure" Reverse Translation approach, we might be justified in ignoring the anomalous cases illustrated above. The RT graphs which would be created for the derived relations in Sequences I-IV, and the actual text generated from them, would still be valid descriptions of their associated relations. The description would be redundant in the first case, and would be a description of an impossible relation in the second case. (However, the description would still be valid. A relation defined as having tuple values which are some value and are not that value simultaneously is an empty relation.)

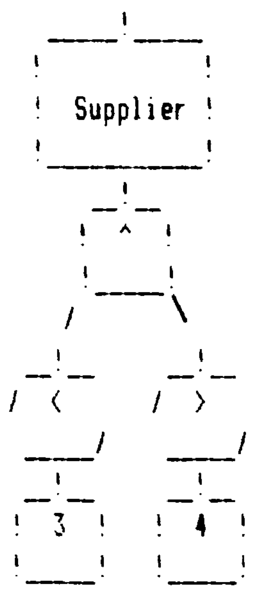


Figure 5.4 An Impossible Entity

5.4.2 Reduction Tables

If the aim of Reverse Translation is to provide validation of queries, to permit the user to formulate the kind of query expressions described in the previous section would be perverse if a solution can be found. The solution to the problems raised in the previous sections is to implement a method whereby the proposed composition of two Simple Comparison Sub-graphs can be checked for redundancy or impossibility. The approach described allows us to simplify Qualification Sub-graphs where the proposed operation is evidently purposeful (as in Sequence I, where a smaller sub-set is being extracted from within a sub-set), and to signal evidently purposeless operations, (as in Sequences Ia - IV, where the result relation is either necessarily empty or identical to one of its parents.)

The method of solution involves implementing the procedure illustrated in Tables 5.1 - 5.4. These record the actions to be taken upon input of all possible variants of two Simple Logical Sub-graph arguments, together with the Logical Node (an AND or OR node) with which it is proposed to compose them. (These tables are actually implemented as a procedure.) The procedure they illustrate must be invoked whenever there is an attempt to attach a Simple Comparison Sub-graph to an existing Qualification Sub-graph. The procedure will return one of the following:

(1) an indication that an error has occured. An attempt to AND the Simple Logical Sub-graphs of Sequences III and IV would yield this result.

(2) a pointer to a Simple Comparison Sub-graph which will be either one of the argument sub-graphs, and one of the three "no change" tokens for "left", "right", "either". This will occur if one of the qualifications is subsumed in the other, or if they are both the same. An attempt to AND the Simple Logical Sub-graphs of Sequences I and II would yield this result. (To anticipate discussion in the next Chapter, it can be noted here that if attaching Sub-graphs to <u>every</u> Principal Entity Node gives the <u>same</u> "no change" token we have a "sub-set error".)

(3) a pointer to a Sub-graph which will be composed from

the Simple Logical Sub-graphs which are input to the procedure as its arguments, and a "new graph" token. This pointer may be to either a Simple Logical Sub-graph composed from the two Simple Comparison Sub-graph arguments, or to a Simple Comparison Sub-graph whose operator token is composed from the operator tokens of its arguments.

The use of these Reduction Tables will prevent the creation of RT graphs with the kind of redundant Qualification Graphs illustrated in Figure 5.3. Not only can Qualification Graphs be reduced to equivalent but more compact forms, but with them we are able to detect queries which would create "impossible" (necessarily empty) or "redundant" (identical to an operand) relations. The next chapter demonstrates the use of these tables in this way.

<u>Key to Tables 5.1 -- 5.4</u>

L: denotes that the left-hand argument graph replaces the proposed composed graph.

R: denotes that the right-hand argument graph replaces the proposed composed graph.

E: either argument graph may replace the proposed composed graph.

N: a Simple Logical Sub-graph replaces the proposed composed graph, with a Comparison Node different from the Comparison Nodes of either argument graph.

error: the proposed composed graph is disjoint.

null: the proposed composed graph is nil.

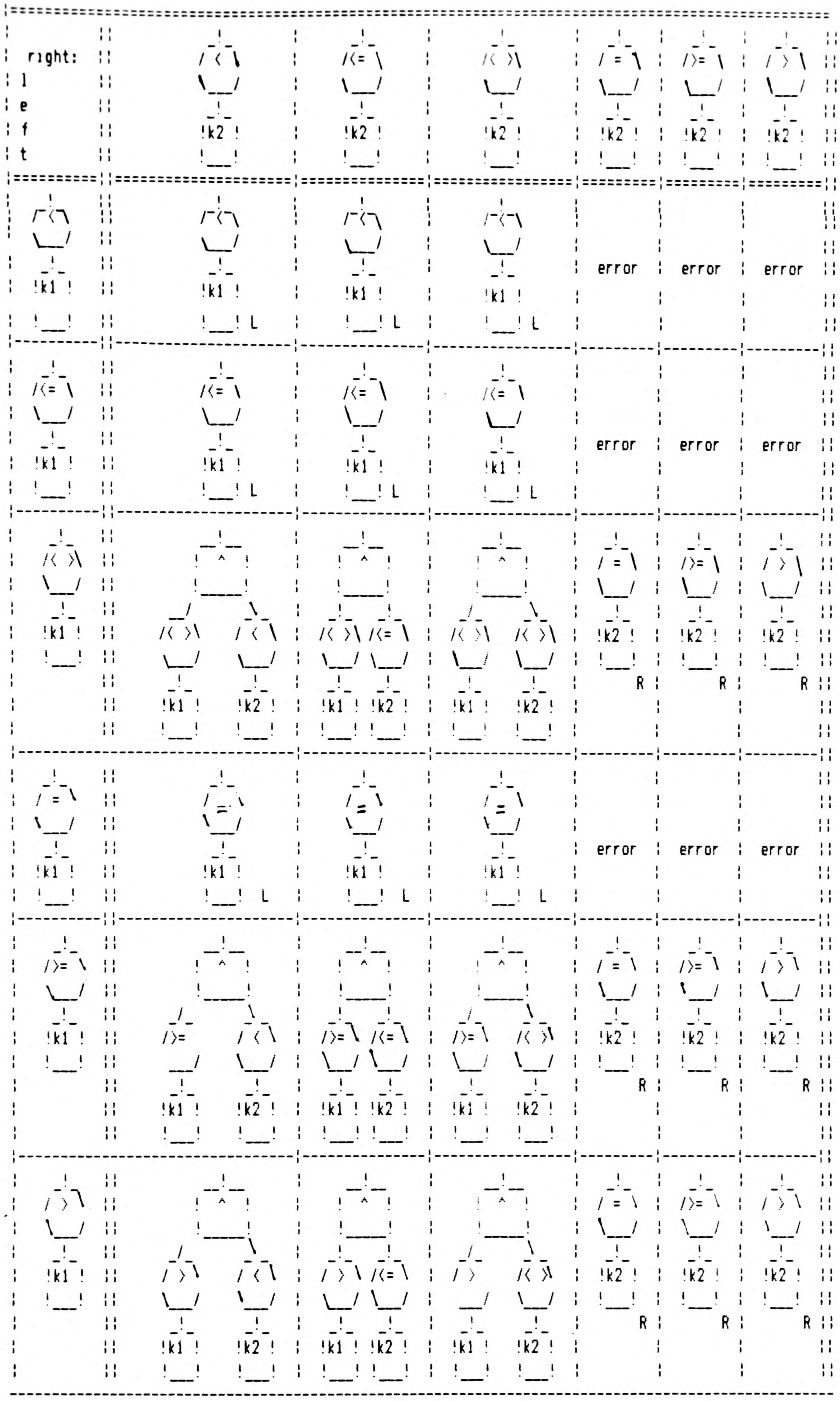


Table 5.1 <u>**Reduction of Intersections, k1 < k2**</u>

```
============================================================================================
!          !!    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !!
! right:   !!   / < \    !   /<= \    !   /< >\    !   / = \    !   />= \    !   / > \    !!
! l        !!   \___/    !   \___/    !   \___/    !   \___/    !   \___/    !   \___/    !!
! e        !!    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !!
! f        !!   !k2 !    !   !k2 !    !   !k2 !    !   !k2 !    !   !k2 !    !   !k2 !    !!
! t        !!   !___!    !   !___!    !   !___!    !   !___!    !   !___!    !   !___!    !!
!==========!!============!============!============!============!============!============!!
!   _!_    !!    _!_     !    _!_     !    _!_     !            !            !            !!
!  / < \   !!   / < \    !   / < \    !   / <      !            !            !            !!
!  \___/   !!   \___/    !   \___/    !    ___/    !            !            !            !!
!   _!_    !!    _!_     !    _!_     !    _!_     !   error    !   error    !   error    !!
!  !k1 !   !!   ! k !    !   !k1 !    !   !k1 !    !            !            !            !!
!  !___!   !!   !___! E  !   !___! L  !   !___! L  !            !            !            !!
!----------!!------------!------------!------------!------------!------------!------------!!
!   _!_    !!    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !            !!
!  /<= \   !!   / < \    !   /<= \    !   / < \    !   / = \    !   / = \    !            !!
!  \___/   !!   \___/    !   \___/    !   \___/    !   \___/    !   \___/    !            !!
!   _!_    !!    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !   error    !!
!  !k1 !   !!   !k2 !    !   ! k !    !   ! k !    !   ! k !    !   ! k !    !            !!
!  !___!   !!   !___! R  !   !___! E  !   !___! N  !   !___! R  !   !___!N   !            !!
!----------!!------------!------------!------------!------------!------------!------------!!
!   _!_    !!    _!_     !    _!_     !    _!_     !            !    _!_     !    _!_     !!
!  /< >\   !!   / < \    !   / < \    !   /< >\    !            !   / > \    !   / > \    !!
!  \___/   !!   \___/    !   \___/    !   \___/    !            !   \___/    !   \___/    !!
!   _!_    !!    _!_     !    _!_     !    _!_     !   error    !    _!_     !    _!_     !!
!  !k1 !   !!   !k2 !    !   ! k !    !   ! k !    !            !   ! k !    !   ! k !    !!
!  !___!   !!   !___! R  !   !___! N  !   !___! E  !            !   !___!N   !   !___!R   !!
!----------!!------------!------------!------------!------------!------------!------------!!
!   _!_    !!            !    _!_     !            !    _!_     !    _!_     !            !!
!  / = \   !!            !   / = \    !            !   / = \    !   / = \    !            !!
!  \___/   !!   error    !   \___/    !   error    !   \___/    !   \___/    !   error    !!
!   _!_    !!            !    _!_     !            !    _!_     !    _!_     !            !!
!  !k1 !   !!            !   !k2 !    !            !   !k  !    !   !k1 !    !            !!
!  !___!   !!            !   !___! L  !            !   !___!E   !   !___!L   !            !!
!----------!!------------!------------!------------!------------!------------!------------!!
!   _!_    !!            !    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !!
!  />= \   !!   error    !   / = \    !   / > \    !   / = \    !   />= \    !   / > \    !!
!  \___/   !!            !   \___/    !   \___/    !   \___/    !   \___/    !   \___/    !!
!   _!_    !!            !    _!_     !    _!_     !    _!_     !    _!_     !    _!_     !!
!  !k1 !   !!            !   ! k !    !   ! k !    !   !k1 !    !   ! k !    !   !k2 !    !!
!  !___!   !!            !   !___! N  !   !___! N  !   !___! R  !   !___!E   !   !___!R   !!
!----------!!------------!------------!------------!------------!------------!------------!!
!   _!_    !!            !            !    _!_     !            !    _!_     !    _!_     !!
!  / > \   !!   error    !   error    !   / > \    !   error    !   / > \    !   / > \    !!
!  \___/   !!            !            !   \___/    !            !   \___/    !   \___/    !!
!   _!_    !!            !            !    _!_     !            !    _!_     !    _!_     !!
!  !k1 !   !!            !            !   !k2 !    !            !   !k2 !    !   ! k !    !!
!  !___!   !!            !            !   !___! L  !            !   !___!N   !   !___!E   !!
--------------------------------------------------------------------------------------------
```

Table 5.2 Reduction of Intersections, k1 = k2

```
==========================================================================================
!          !!    _!_     !    _!_     !    _!_     !      _!_       !      _!_       !      _!_      !!
! right:   !!   / < \    !   /<= \    !   /< >\    !     / = \      !     />= \      !     / > \     !!
! l        !!   \___/    !   \___/    !   \___/    !     \___/      !     \___/      !     \___/     !!
! e        !!    _!_     !    _!_     !    _!_     !      _!_       !      _!_       !      _!_      !!
! f        !!   !k2 !    !   !k2 !    !   !k2 !    !     !k2 !      !     !k2 !      !     !k2 !     !!
! t        !!   !___!    !   !___!    !   !___!    !     !___!      !     !___!      !     !___!     !!
==========!!============!============!============!================!================!===============!!
!    _!_   !!    _!_     !    _!_     !    _!_     !      _!_       !      _!_       !      _!_      !!
!   /<  \  !!   / < \    !   /<= \    !   /< >\    !     ! U !      !     ! U !      !     ! U !     !!
!   \___/  !!   \___/    !   \___/    !   \___/    !     !___!      !     !___!      !     !___!     !!
!    _!_   !!    _!_     !    _!_     !    _!_     !    _/   \_     !    _/   \_     !    _/   \_    !!
!   !k1 !  !!   !k2 !    !   !k2 !    !   !k2 !    !  / < \ / = \   !  / < \ / >=\   !  / < \ / > \  !!
!   !___!  !!   !___! R  !   !___! R  !   !___! R  !  \___/ \___/   !  \___/ \___/   !  \___/ \___/  !!
!          !!            !            !            !   _!_   _!_    !   _!_   _!_    !   _!_   _!_   !!
!          !!            !            !            !  !k1 ! !k2 !   !  !k1 ! !k2 !   !  !k1 ! !k2 !  !!
!          !!            !            !            !  !___! !___!   !  !___! !___!   !  !___! !___!  !!
!----------!!------------!------------!------------!----------------!----------------!---------------!!
!    _!_   !!    _!_     !    _!_     !    _!_     !      _!_       !      _!_       !      _!_      !!
!   /<= \  !!   / < \    !   /<= \    !   /< >\    !     ! U !      !     ! U !      !     ! U !     !!
!   \___/  !!   \___/    !   \___/    !   \___/    !     !___!      !     !___!      !     !___!     !!
!    _!_   !!    _!_     !    _!_     !    _!_     !    _/   \_     !    _/   \_     !    _/   \_    !!
!   !k1 !  !!   !k2 !    !   !k2 !    !   !k2 !    !  /<=   / = \   !  /<= \ />= \   !  /<= \ / > \  !!
!   !___!  !!   !___! R  !   !___! R  !   !___! R  !    __/ \___/   !  \___/ \___/   !  \___/ \___/  !!
!          !!            !            !            !   _!_   _!_    !   _!_   _!_    !   _!_   _!_   !!
!          !!            !            !            !  !k1 ! !k2 !   !  !k1 ! !k2 !   !  !k1 ! !k2 !  !!
!          !!            !            !            !  !___! !___!   !  !___! !___!   !  !___! !___!  !!
!----------!!------------!------------!------------!----------------!----------------!---------------!!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   /< >\  !!   null     !   null     !   null     !     /< >\      !     /< >\      !     /< >\     !!
!   \___/  !!            !            !            !     \___/      !     \___/      !     \___/     !!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   !k1 !  !!            !            !            !     !k1 !      !     !k1 !      !     !k1 !     !!
!   !___!  !!            !            !            !     !___! L    !     !___! L    !     !___! L   !!
!----------!!------------!------------!------------!----------------!----------------!---------------!!
!    _!_   !!    _!_     !    _!_     !    _!_     !      _!_       !      _!_       !      _!_      !!
!   / = \  !!   / < \    !   /<= \    !   /< >\    !     ! U !      !     ! U !      !     ! U !     !!
!   \___/  !!   \___/    !   \___/    !   \___/    !     !___!      !     !___!      !     !___!     !!
!    _!_   !!    _!_     !    _!_     !    _!_     !    _/   \_     !    _/   \_     !    _/   \_    !!
!   !k1 !  !!   !k2 !    !   !k2 !    !   !k2 !    !  / = \ / = \   !  / = \ />= \   !  / = \ / > \  !!
!   !___!  !!   !___! R  !   !___! R  !   !___! R  !  \___/ \___/   !  \___/ \___/   !  \___/ \___/  !!
!          !!            !            !            !   _!_   _!_    !   _!_   _!_    !   _!_   _!_   !!
!          !!            !            !            !  !k1 ! !k2 !   !  !k1 ! !k2 !   !  !k1 ! !k2 !  !!
!          !!            !            !            !  !___! !___!   !  !___! !___!   !  !___! !___!  !!
!----------!!------------!------------!------------!----------------!----------------!---------------!!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   />= \  !!   null     !   null     !   null     !     />= \      !     />= \      !     / > \     !!
!   \___/  !!            !            !            !     \___/      !     \___/      !     \___/     !!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   !k1 !  !!            !            !            !     !k1 !      !     !k1 !      !     !k1 !     !!
!   !___!  !!            !            !            !     !___! L    !     !___! L    !     !___! L   !!
!----------!!------------!------------!------------!----------------!----------------!---------------!!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   / > \  !!   null     !   null     !   null     !     / > \      !     / > \      !     / > \     !!
!   \___/  !!            !            !            !     \___/      !     \___/      !     \___/     !!
!    _!_   !!            !            !            !      _!_       !      _!_       !      _!_      !!
!   !k1 !  !!            !            !            !     !k1 !      !     !k1 !      !     !k1 !     !!
!   !___!  !!            !            !            !     !___! L    !     !___! L    !     !___! L   !!
------------------------------------------------------------------------------------------
```

Table 5.3 **Reduction of unions, k1 < k2**

```
=====================================================================================================
|           ||        _!_         |      _!_      |       _!_       |   _!_   |   _!_   |   _!_   ||
| right:    ||       / < \        |     /<= \     |      /< >\      |  / = \  |  />= \  |  / > \  ||
| l         ||       \___/        |     \___/     |      \___/      |  \___/  |  \___/  |  \___/  ||
| e         ||        _!_         |      _!_      |       _!_       |   _!_   |   _!_   |   _!_   ||
| f         ||       !k2 !        |     !k2 !     |      !k2 !      |  !k2 !  |  !k2 !  |  !k2 !  ||
| t         ||       !___!        |     !___!     |      !___!      |  !___!  |  !___!  |  !___!  ||
===========||====================|===============|=================|=========|=========|=========||
|    _!_    ||        _!_         |      _!_      |       _!_       |   _!_   |         |   _!_   ||
|   / < \   ||       / < \        |     /<= \     |      /< >\      |  /<= \  |  null   |  /< >\  ||
|   \___/   ||       \___/        |     \___/     |      \___/      |  \___/  |         |  \___/  ||
|    _!_    ||        _!_         |      _!_      |       _!_       |   _!_   |         |   _!_   ||
|   !k1 !   ||       !k2 !        |     !k2 !     |      !k2 !      |  ! k !  |         |  ! k !  ||
|   !___!   ||       !___! R      |     !___! R   |      !___! R    |  !___! N|         |  !___! N||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
|    _!_    ||        _!_         |      _!_      |                 |   _!_   |         |         ||
|   /<= \   ||       /<= \        |     /<= \     |      null       |  /<= \  |  null   |  null   ||
|   \___/   ||       \___/        |     \___/     |                 |  \___/  |         |         ||
|    _!_    ||        _!_         |      _!_      |                 |   _!_   |         |         ||
|   !k1 !   ||       !k1 !        |     ! k !     |                 |  ! k !  |         |         ||
|   !___!   ||       !___! R      |     !___! E   |                 |  !___! R|         |         ||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
|    _!_    ||        _!_         |               |       _!_       |         |         |   _!_   ||
|   /< >\   ||       /< >\        |     null      |      /< >\      |  null   |  null   |  /< >\  ||
|   \___/   ||       \___/        |               |      \___/      |         |         |  \___/  ||
|    _!_    ||        _!_         |               |       _!_       |         |         |   _!_   ||
|   !k1 !   ||       !k1 !        |               |      ! k !      |         |         |  !k1 !  ||
|   !___!   ||       !___! R      |               |      !___! E    |         |         |  !___!R ||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
|    _!_    ||        _!_         |      _!_      |                 |   _!_   |   _!_   |   _!_   ||
|   / = \   ||       /<= \        |     /<= \     |      null       |  / = \  |  />= \  |  />= \  ||
|   \___/   ||       \___/        |     \___/     |                 |  \___/  |  \___/  |  \___/  ||
|    _!_    ||        _!_         |      _!_      |                 |   _!_   |   _!_   |   _!_   ||
|   !k1 !   ||       !k  !        |     !k2 !     |                 |  ! k !  |  !k2 !  |  !k  !  ||
|   !___!   ||       !___! N      |     !___! R   |                 |  !___! E|  !___!R |  !___!N ||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
|    _!_    ||                    |               |                 |   _!_   |   _!_   |   _!_   ||
|   />= \   ||        null        |     null      |      null       |  />=    |  />=    |  />= \  ||
|   \___/   ||                    |               |                 |    __/  |    __/  |  \___/  ||
|    _!_    ||                    |               |                 |   _!_   |   _!_   |   _!_   ||
|   !k1 !   ||                    |               |                 |  !k2 !  |  ! k !  |  !k2 !  ||
|   !___!   ||                    |               |                 |  !___! R|  !___!E |  !___!R ||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
|    _!_    ||        _!_         |               |       _!_       |   _!_   |   _!_   |   _!_   ||
|   / > \   ||       /< >\        |     null      |      /< >\      |  />= \  |  />= \  |  / > \  ||
|   \___/   ||       \___/        |               |      \___/      |  \___/  |  \___/  |  \___/  ||
|    _!_    ||        _!_         |               |       _!_       |   _!_   |   _!_   |   _!_   ||
|   !k1 !   ||       ! k !        |               |      !k2 !      |  ! k !  |  !k2 !  |  !k1 !  ||
|   !___!   ||       !___! N      |               |      !___! R    |  !___! N|  !___!R |  !___! L||
|-----------||--------------------|---------------|-----------------|---------|---------|---------||
```

Table 5.4 <u>Reduction of Unions, k1 = k2</u>

5.4.3 Attaching a Simple Comparison Sub-graph to a Complex Qualification Sub-Graph

Previous examples in this chapter have assumed that, at most, an Entity Node prior to its final processing is qualified with a Simple Comparison Sub-graph (a single Comparison Node pointing to a single Value Node). Now the case of Complex Qualification Sub-graphs is considered.

If a Complex Qualification Sub-graph already exists, then its root node is either an OR Logical Node or an AND Logical Node. If the root node is an OR, then this algorithm is invoked again for both of this node's left and right descendants. This algorithm guarantees that if a Qualification Sub-graph has OR nodes, they will always occur above all other nodes. It also implies that whenever an OR node is encountered, then both the descendant sub-trees of the OR node will have the new Simple Comparison Sub-graph attached as a component (although the resulting Sub-graph may be reduced). This is equivalent to the distributive logical identity,

(A OR B) AND C <=> (A AND C) OR (B AND C)

If the node pointed to is an AND Logical Node, then if the Comparison Node of the Simple Comparison Sub-graph to be attached is a token for the ">" or ">=" comparators, then descend to the AND node's left leaf Simple Comparison Sub-graph, reduce the left leaf Simple Comparison Sub-graph with the new Simple Comparison Sub-graph, and reduce the result (which will be a Simple Comparison

Graph) again with the AND node's right leaf Simple Comparison Sub-graph.

If the Comparison operator is "<" or "<=", apply this procedure to the right leaf Simple Comparison Sub-graph.

If the Comparison operator is the equality operator, apply the procedure to both leaves, which will eliminate them both. Then apply it to the results, which will eliminate one of them. This is equivalent to replacing the AND Logical Node with the new Simple Comparison Sub-graph.

If the Comparison operator of the Simple Logical Sub-graph to be added is the inequality operator, then if its leaf value is greater than or equal to the leaf value of the AND node's right descendant Simple Comparison Sub-graph, or less than or equal to the value of the left descendant Simple Comparison Sub-graph, then reduce it with the appropriate one of these Sub-graphs (to allow the reduction procedure to generate a warning or error message). Otherwise, create a new AND node pointed to by the immediate ancestor AND node. Let the Simple Comparison Sub-graph be the new AND node's right descendant and let the original AND node's left descendant be the new AND node's left descendant. If the left descendant of the new AND node is a Simple Comparison Sub-graph, then stop. Otherwise (the left desecendant is an AND node), compare the current AND node's right descendant value with the right descendant value of its left descendant AND node, and put into lexicographical order by swapping value nodes if necessary. If the value nodes hold identical values eliminate the current AND node and its right descendant and issue a "redundant operation" message. This guarantees that if there is an "AND node chain" the chain will always have Simple Comparison Sub-Graphs as descendants of its right-hand pointers and AND nodes as descendants of its left-hand pointers, except for the final leaf. The Value Nodes of such a chain will be in lexicographical order. Such a chain will only be the case if there are Comparison Nodes holding inequality operator tokens, since otherwise the Query Graph reduction procedure will always reduce the graph to a Simple Logical Sub-Graph or Simple Comparison Sub-Graph.

5.4.4 Attaching a Simple Logical Sub-graph to an Entity Node with a Complex Qualification Sub-graph

```
SPJ3  <-  SPJ2  : [PNO < > 3 ]
```

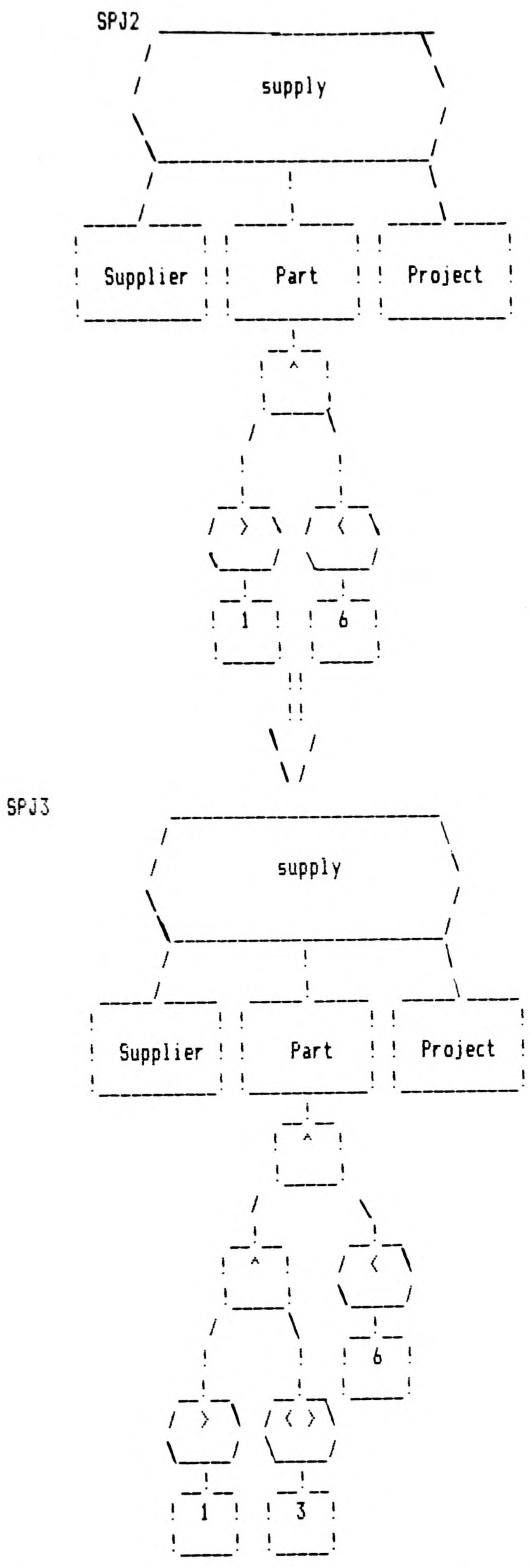


Figure 5.5 <u>The effect of selection on an Entity Node with a Complex Qualification Sub-graph</u>

5.4.5 Transferring Complex Qualification Sub-graphs

The next chapter demonstrates the necessity for transferring a Complex Qualification Sub-graph from one Entity Node to another Entity Node which may already have its own Complex Qualification Sub-graph. This transferral is accomplished by decomposing the graph to be transferred into one Simple Comparison Sub-graph at a time, and then attaching it to the target Entity Node using the algorithm given in Section 5.4.3. Each Simple Comparison Sub-graph taken from the Complex Qualification Sub-graph being transferred results in the contraction of the graph through loss of the transferred sub-graph and its AND node.

The "before" and "after" positions are illustrated in Figure 5.6.

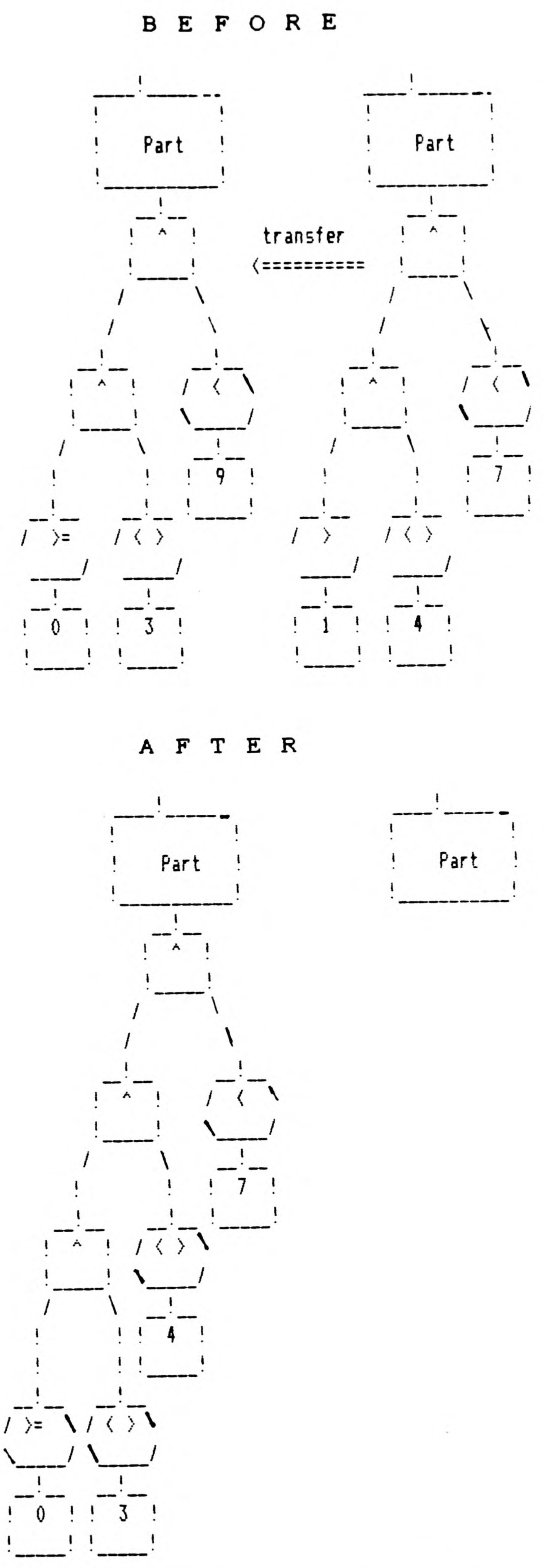


Figure 5.6 Transferral of a Complex Qualification Sub-graph

CHAPTER SIX

SET OPERATIONS

6.1 Introduction

Chapter Four described the Reverse Translation approach to generating Reverse Translation graphs for derived relations. Only Base Relations were used to illustrate the effect of each relational operation. The concepts advanced in that chapter were:

(1) PROJECTION is represented in a graph by setting the attribute reference of the relevant Entity Nodes to zero.
(2) SELECTION is represented by attaching a Qualification Sub-graph to the relevant Entity Node. Chapter Five extended the work on SELECTION by demonstrating how repeated SELECTIONs built a Sub-graph tree, whose construction required the possible simplification of Sub-graphs and perhaps the flagging of semantic errors.
(3) UNION is represented by making the operand graphs the descendants of an OR Logical Node.
(4) DIFFERENCE is represented by making the operand graphs the descendants of a NOT Logical Node, and connecting corresponding Entity Nodes in an Entity Node Chain.
(5) INTERSECTION and JOIN are represented by making the operand graphs the descendants of an AND Logical Node, and connecting corresponding Entity Nodes (if any) in an Entity Node Chain.

These operations may be divided into two classes from the viewpoint of Reverse Translation: (1) those operations which take as an argument a single relation (PROJECTION, PERMUTATION and SELECTION), and (2) those which take as arguments two relations (UNION, INTERSECTION, DIFFERENCE and JOIN). An operation of the first class generates an RT graph which is a simple modification of the graph of its parent relation and which requires no further transformation beyond the measures described in Chapter Four, and in the case of SELECTION, a test/reduction of the selected-on attribute's corresponding Principal Entity Node's Qualification Graph as described in Chapter Five. An operation of the second class is a more complex matter, since it produces a derived relation whose graph must, to reflect the semantics of its parent relation, be composed from the graphs of the operand relations. This chapter takes up the creation, and possible reduction, of RT graphs whose parent graphs may be Complex RT graphs.

The three set operations UNION, DIFFERENCE and INTERSECTION are considered first. The discussion of each particular operation is preceded with an explanation of the treatment that is applicable to any set operation, to trap queries which must signal user confusion. Chapter Five described a mechanism for reducing redundant Qualification Sub-graphs and also checking for "no-change" or "necessarily empty" result relations. In that chapter, this mechanism was applied to cases of repeated SELECTION on a single relation. In this chapter the same mechanism is extended to cover each of the set operations.

6.2 Predication-identical Graphs

6.2.1 Terminology introduced in this Chapter

Two graphs are predication identical if and only if they are identical down to Entity Node level. This means they are predication-identical if each corresponding pair of Predication Nodes are the same permutation and the same sense of the same predication, if each Predication Node has the same configuration of present and absent Entity Nodes and if each corresponding pair of Entity Nodes has the same entity identifier. (The effect of the latter requirement is to ensure that each graph has the same configuration of Entity Node Chains.)

6.2.2 Examples

6.2.2.1 Examples of Relations with Non-Predication-Identical Graphs

Assume P1 is a relation of degree two, with both attributes (PNO1 and PNO2) drawn from the domain of Parts, related by the predication "includes" in the PNO1 to PNO2 direction and "is included in" in the PNO2 to PNO1 direction. Then let

```
        P2  <-  P1  # [PNO2, PNO1]
```

The graphs of P1 and P2 meet every criterion for predication-identicality except for their not being of the same permutation. Let

```
     P1A  <-  P1  % [PNO1]    P1B  <-  P1  % [PNO2]
```

The graphs of P1A and P2A meet every criterion for predication-identicality except for not having the same configuration of present and absent Entity Nodes. (The PROJECTION operation on each relation has projected-out non-corresponding Entity Nodes.)

6.2.2.2 Examples of Relations with Predication-Identical Graphs

```
     P3  <-  P1  :  [ PNO1 > 10 ]
     P4  <-  P1  :  [ PNO2 =  5 ]
     P5  <-  P4  :  [ PNO1 < 20 ]
```

P1, P3, P4, P5 are all predication-identical.

```
     P6 <- P3 - P5
     P7 <- P1 - P4
```

P6 and P7 are predication-identical.

6.2.3 The Significance of Predication-Identicality

Two predication-identical graphs may be reducable to the left-hand graph augmented with the (possibly transformed)

Qualification Sub-graphs of the right-hand graph.

6.3 <u>Trapping Errors in Set Operations on Relations known to conform to certain constraints</u>

6.3.1 <u>Known set relationships among relations</u>

In general, we may note that some set operations can be syntactically valid but yield a result which is tautological, that is, which contains no information not present before the operation. Any operation which necessarily yields an empty relation, or a relation which is identical to one of the operand relations, should be identified and reported as such to the user. (Reverse Translation as such is not the only possible method for signalling the existence of an invalid query in these cases However, the techniques used for implementing Reverse Translation lend themselves easily to use in flagging such invalid queries.) Table 6.3 records semantically invalid operations on relations for which it is known (either directly or by deduction) that the following relationships apply:

A:	R2 is a proper subset of R1.
B:	R1 is a proper subset of R2.
C:	R1 and R2 are disjoint.
D:	R1 and R2 intersect.
E:	R1 is identical to R2.

Table 6.1 <u>Possible relationships between any two set-operation compatible relations R1 and R2</u>

6.3.2 Valid and Invalid Operations on Sets With Known Relationships

The validity of these five cases under the three set operations is recorded in the following table, where "V" means (potentially) valid and a number refers to a footnote explaining why this operation is a tautology:

Case	R1 U R2	R1 ^ R2	R1 - R2
A	1	2	V
B	2	1	3
C	V	3	3
D	V	V	V
E	1/2	1/2	3

1: Yields R1.
2: Yields R2.
3: Yields a relation with cardinality zero.

Table 6.3 Tautological set operations between two relations, R1 and R2

6.4 Tests Applicable to Set Operations

6.4.1 Testing for Relation Identity

We must first check to see that the user is not dealing with two identical relations, by comparing their current graphs using the function "Compare" [Appendix, Program Listing.] This function takes as a parameter two pointers to RT graphs and compares the graphs pointed to on a node-by-node basis, returning TRUE if the graphs are identical in every respect. If this occurs, an appropriate error message is generated. (This is Case "E" of Table 6.1.) If we chose not to distinguish between cases of set operations between identical relations and maladroit attempts to combine relations, one of which is a proper sub-set of the other, this comparison for graph identity could be dispensed with since identical relation errors would be caught by the sub-set test.

6.4.2 Testing for inclusion and disjointness

6.4.2.1 General Algorithm

Assuming that the query passes the "identity test" of the previous section the following algorithm can be applied:

```
For each of the right-hand graph's Entity Nodes do the
following:
  IF it corresponds to an existing attribute,
  THEN
     IF it has a Qualification Sub-graph
     THEN
        Copy its Qualification Sub-graph

        Copy the Qualification Sub-graph of the
        Node which would be its Principal Entity Node were
        they to be linked in an Entity Node Chain.

        Attempt to merge them using the algorithm
        described in Chapter Five.

        Keep count of the number of left graph, right
        graph, new graph, and error conditions returned.
```

6.4.3.2 Testing for disjointness

If one or more errors were returned during the course of Qualification Sub-graph transfer, then the two relations are disjoint. (Case "C" of Table 5.2) (If an RT graph includes OR Logical Nodes above Predication Nodes, then the disjoint test is failed only if both descendants of the OR node return disjoint indicators.) Unlike subset errors, disjoint errors can be detected whether or not the graphs of operand relations are predication-identical. If we know that all values of the attribute of the first relation in the example mentioned above are less than ten, and all corresponding values in the second relation are greater than ten, then we do know that the attributes in question have no values in common, making the relations disjoint.

6.4.3.3 Testing for set inclusion

If the two operand RT graphs are predication-identical, the count of the number of "no change" values returned while transferring the right-hand graph's Qualification Sub-graphs to the Principal Entity Nodes of the left-hand graph is significant: if after the entire operation, there were zero changes, then one operand is a sub-set of the other. (Cases "A" or "B" of Table 6.1.) If the two relations are not predication-identical, then it is not possible to test whether one was necessarily a sub-set of the other, within the terms of the data model used in this thesis.

For example, consider two relations, not predication-identical. Assume that, for each relation, only one attribute has been the object of a previous SELECTION. One relation might have an attribute for which is recorded in its corresponding Entity's Qualification Sub-graph, created via a previous SELECTION, that all its values are less than 10. The second relation's equivalent attribute might have a similar Qualification Sub-graph recording values of less than 5. But the second relation could have values in its attribute not present in the first. Therefore it is not necessarily a sub-set of the first.

If set relationships among relations were among the constraints enforced and recorded, then it would be possible to carry out the sub-set test on those relations known, via the system schema, to be identical. For example, if we knew that a relation SP % [SNO] was identical to a relation SD % [SNO], then sequences of SELECTIONS on these relations followed by INTERSECTIONS, JOINS or DIFFERENCES could be put to the sub-set test.

6.5 Processing an INTERSECTION

6.5.1 Predication identical and non-identical graphs

The derived graph of an INTERSECTION operation which has passed the tests for identity, the sub-set condition (if applicable) and disjointness will be composed of the two operand relation's graphs, in the following way:

Create an AND Logical Node and make it the root node of the derived relation's graph. Create a copy of the right-hand relation's graph and make it the right descendant of the AND node. Make a copy of the left-hand relation's graph and make it the left descendant. Link all Equivalent Entity Nodes in an Entity Node Chain. Transfer all Qualification Sub-graphs from the right-hand graph to their Principal Entity Nodes. If the right hand graph is not predication-identical to the left-hand graph, then stop. (If predication-identicallity does obtain between the two descendant graphs, carry out the operations spelled out in Section 6.5.2.) The new graph will be composed of the two operand graphs as descendants of an AND Logical Node, all corresponding Entity Nodes linked in an Entity Node Chain, with the Qualification Sub-graphs of the of Entity Nodes representing non-projected out attributes in the right-hand graph transferred to their Principal Entity Node and dropped from the contributing Equivalent Entity Node.

Such transferral for Entity Nodes linked in an Entity Node Chain is justified on the grounds that all of the Entity Nodes in a particular chain correspond to the same attribute, and thus bear information about every value of that attribute in a tuple of its relation. In a chain that has been created by an INTERSECTION or JOIN operation this information is simultaneously true for any value of that attribute. Put another way, whatever can be said about one Entity Node in such a chain can be said about every other Entity Node in the chain. Thus no information is lost by putting all Qualification Sub-graphs on one Entity Node in the chain. The positive justification for doing so is to allow the Reverse Translation to give all of the "qualifying" information held by Qualification Sub-graphs at the first encounter with a particular entity's Entity Node chain, rather than dispersing this information throughout the Reverse Translation.

6.5.2 Graph Reduction Following Intersection

The INTERSECTION of two predication-identical graphs can always be reduced, as follows.

(1) Carry out the algorithm for INTERSECTION, above.

(2) Now transfer any remaining Qualification Sub-graphs of projected-out Entity Nodes in the right hand graph to their corresponding Entity Nodes in the left hand graph. (The two graphs are identical down to Entity Node level, so this may be accomplished by visiting each Entity Node in the right hand graph in tandem with an identical traversal of the left hand graph. Where a projected-out Entity Node has a Qualification Sub-graph, this is transferred to its corresponding Entity Node in left hand graph. The attachment is via an AND Logical Node which becomes the root node of the left hand Entity Node's Qualification Sub-graph, with this Entity Node's previous Qualification Sub-graph as the left descendant and the transferred Qualification Sub-graph as the right

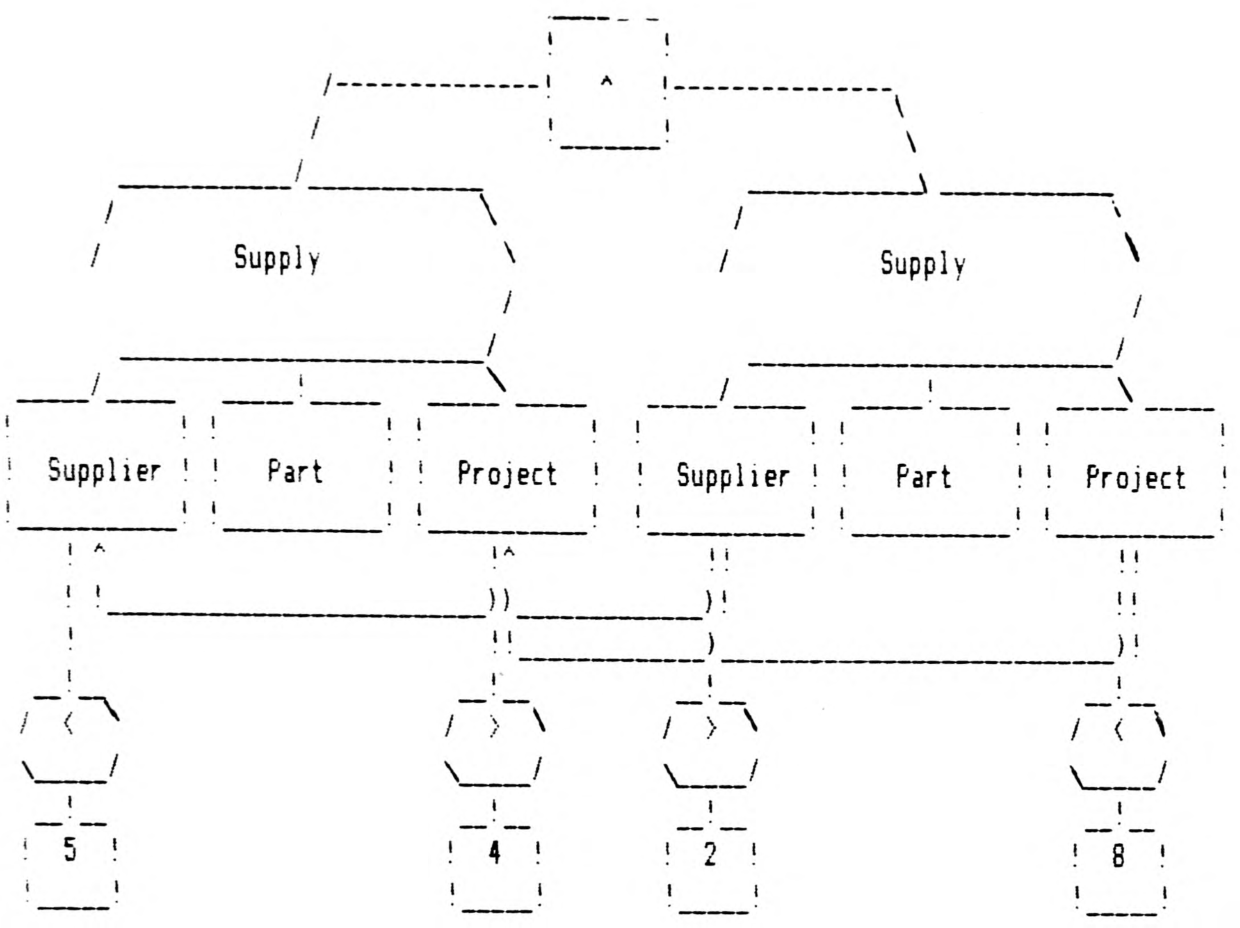
descendant. The two sub-graphs are reduced as for the transferral of Qualification Sub-graphs between non-projected out Entity Nodes, except that the tables for the UNION operation are used instead of the tables for the INTERSECTION operation. (The Logical Node linking reduced graphs remains an AND Logical Node, however.) The justification for this is given in Section 6.5.3.

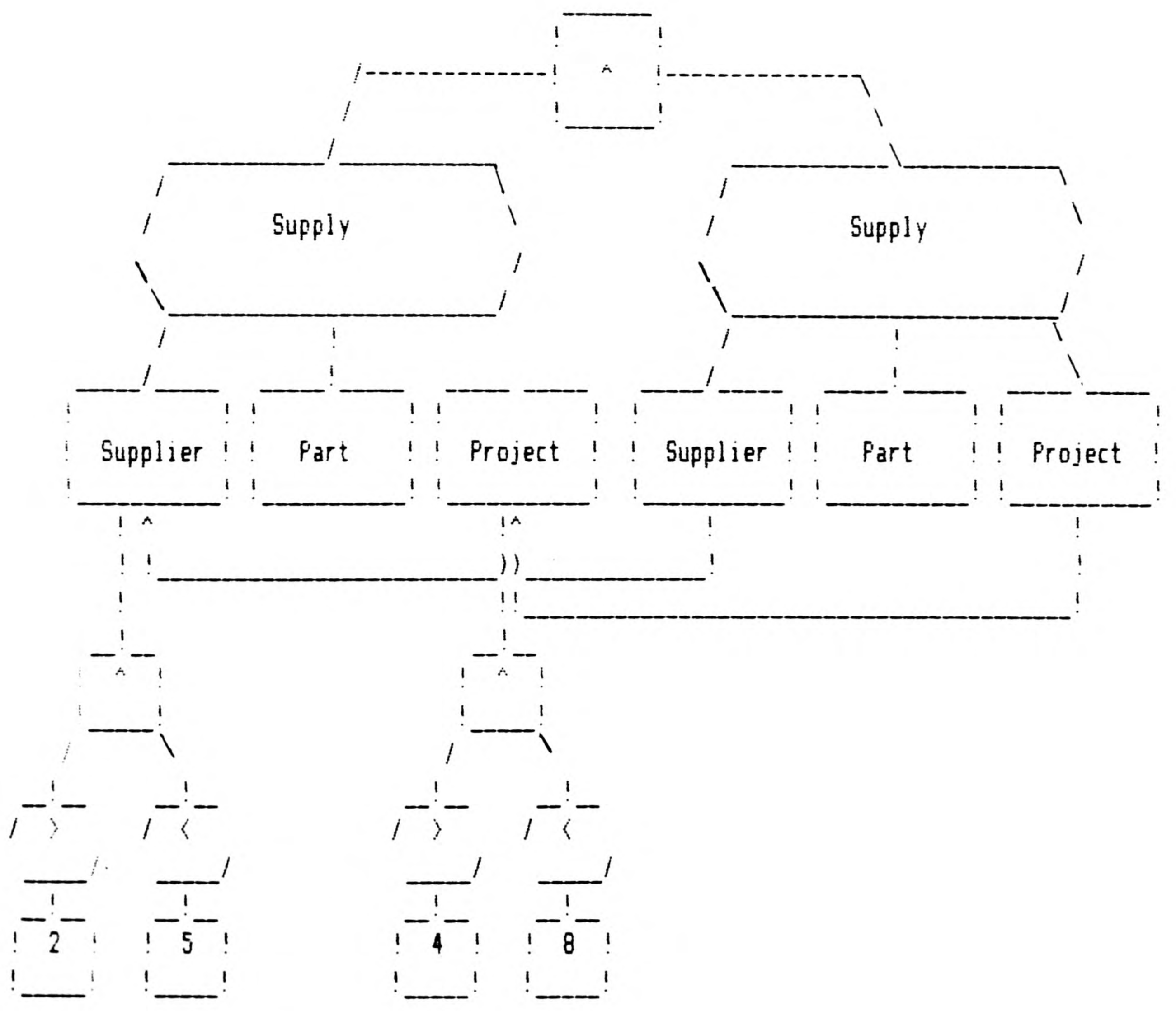
(3) Delete the root AND Logical Node and the right hand graph, and make the augmented left hand graph's root node the root node of the relation.

6.5.3 Attaching Qualification Sub-graphs to Entity Nodes of projected-out attributes

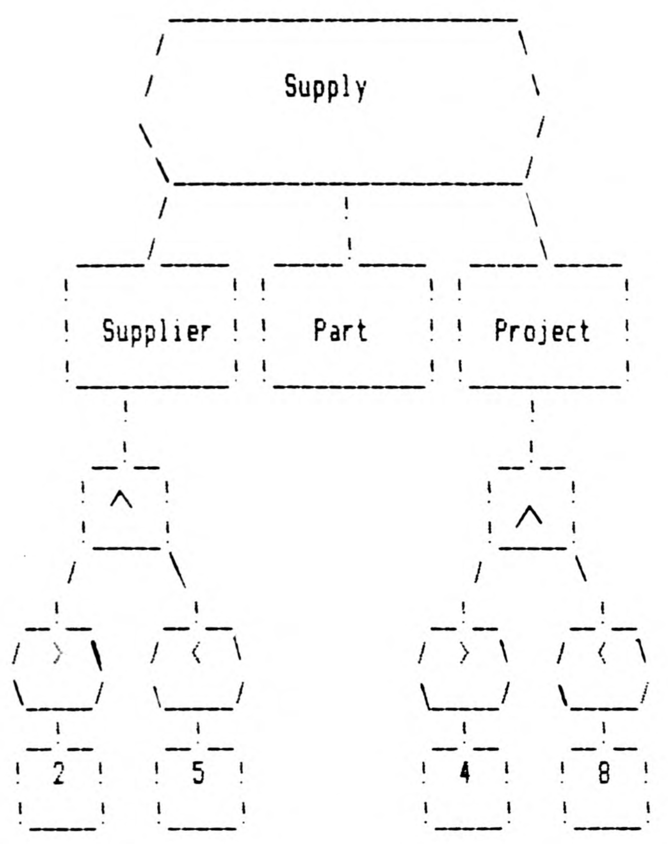
An Entity Node of an attribute which has not been projected out refers to an existing value. It tells the range within which this value can lie, and possibly what values it is not. An Entity Node of a projected-out attribute, unless it is of Edegree 1, refers to many possible values. The logic of set union applies.

Reduction of predication-identical graphs with no projected-out nodes. Following the algorithm for INTERSECTION, first the corresponding Entity Nodes are linked. Then the Qualification Sub-graphs of the right-hand graph are transferred to the Principal Entity Nodes of the left-hand graph.


^
Supply
Supply
Supplier
Part
Project
Supplier
Part
Project
<
>
>
<
5
4
2
8


^
Supply
Supply
Supplier
Part
Project
Supplier
Part
Project
^
^
>
<
>
<
2
5
4
8

As the two graphs are predication-identical, they may now be reduced to:



CHAPTER SEVEN

APPLICATIONS OF REVERSE TRANSLATION

7.1 Introduction

In this chapter the Reverse Translation method is applied to several situations which can constitute "traps" for the naive user. The first trap considered is the well-known "Connection Trap", whose existence was documented simultaneously with the original formulation of the relational model. Then a family of traps which will be termed "commutation traps", for predications of degrees two and three is examined.

7.2 The Connection Trap

The "connection trap" was first discussed by EF Codd in the paper which introduced the relational model to the computing community [Codd, 1970]. "Connection traps" can arise when performing JOIN operations on relations which are themselves projections of other relations. In this situation, a resulting relation may have the "surface structure" of the base relation, but contain spurious tuples not in the original. The following example illustrates the problem.

Given the relation

SPJ

SNO	PNO	JNO
[SUPPLIER]	[PART]	[PROJECT]
1	1	1
1	1	2
2	1	1

with the meaning that Supplier SNO supplies Part-type PNO to Project JNO, let

```
S1  <--  SPJ %[SNO, PNO]
S2  <--  SPJ %[PNO, JNO]
```

then

S1

SNO	PNO
[SUPPLIER]	[PART]
1	1
2	1

(The meaning of S1 is that Supplier SNO supplies Part-type PNO to one or more projects.)

and

S2

PNO	JNO
[PART]	[PROJECT]
1	1
1	2

(The meaning of S2 is that Part-type PNO is supplied to Project JNO by one or more Suppliers.) The connection trap can arise if S1 and S2 are re-joined, with PNO as the common attribute.

S3 <-- S1 * S2 (Join over PNO.)

S3

SNO [SUPPLIER]	PNO [PART]	JNO [PROJECT]
1	1	1
1	1	2
2	1	1
2	1	2

The last tuple did not exist in the original relation. Its appearance here is an artefact of the semantics of the JOIN operation. The application of Reverse Translation graphs to generate descriptions of the relations provides a way to avoid confusing S3 with SPJ:

SPJ

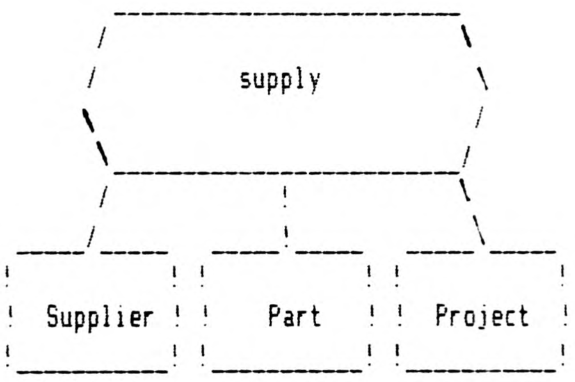


Figure 7.1 The graph of SPJ

S1

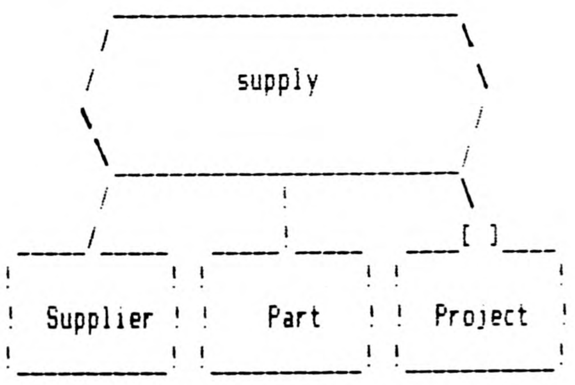


Figure 7.2 The graph of S1

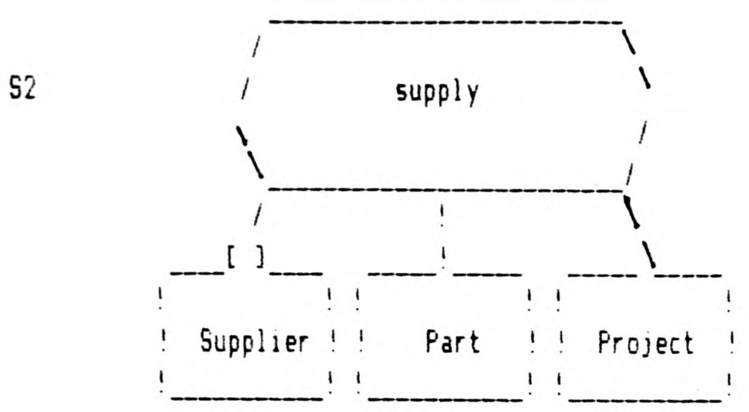


Figure 7.3 The graph of S2

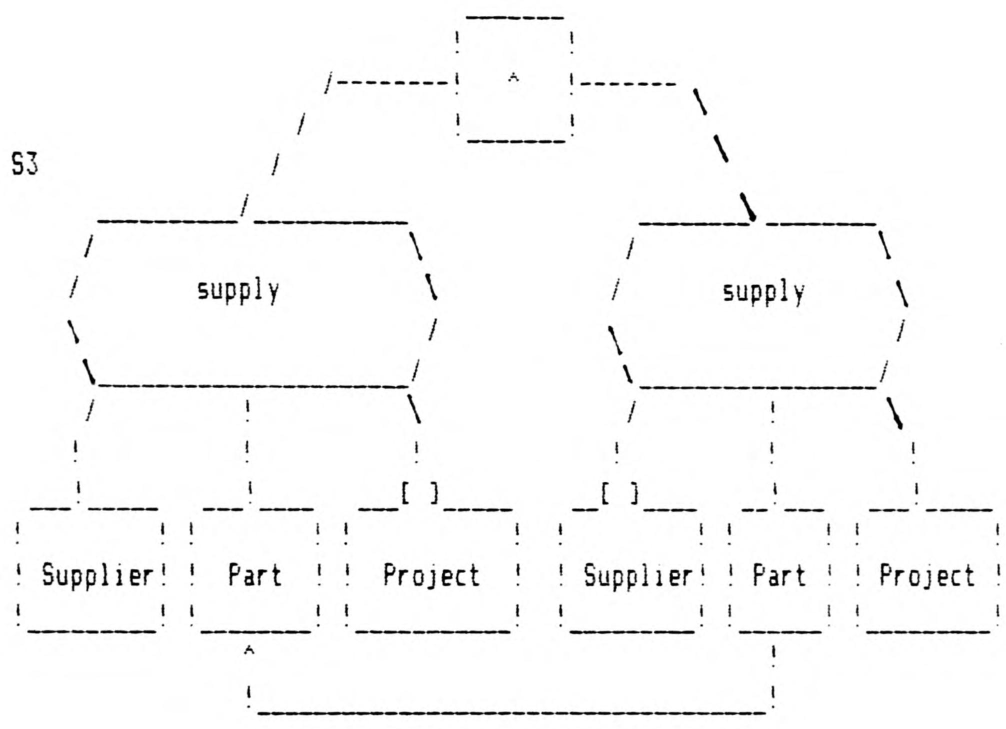


Figure 7.4. The graph of S3

The semantics of SPJ are clearly visible in Figure 7.1. The entities within a given tuple are clearly bound up in a single, indivisible relationship. The effect of projection on SPJ is also clearly seen in Figure 7.2 and Figure 7.3, and the effect of the JOIN in creating a relation with a quite different meaning from SPJ is illustrated by the graph shown in Figure 7.4, which contrasts clearly with the graph of SPJ shown in Figure 7.1.

The actual Reverse Translations which will be generated when the two RT graphs are processed likewise stand in sharp contrast to each other.

The Reverse Translation of SPJ:

```
The indicated Supplier [SNO], and possibly other Suppliers,
supplies
the indicated Part [PNO], and possibly other Parts,
to
the indicated Project [JNO], and possibly other Projects.
```

The Reverse Translation of S3:

```
The indicated Supplier [SNO], and possibly others,
supplies
the indicated Part [PNO], and possibly others,
to
at least one Project
and
At least one Supplier
supplies
the indicated Part [PNO], and possibly other Parts,
to
the indicated Project [JNO], and possibly other Projects.
```

7.3 The Selection Trap

7.3.1 Background

Another possible trap due to user misunderstanding of the semantics of query expressions can arise when the SELECTION operation is invoked to extract all attribute values in an N:M relationship which do not take part in the relationship with a designated value of another attribute. This might be conjectured to be particularly liable to occur if the selection is followed by a PROJECTION prior to display of the final result, since, as in the "connection trap", the "surface structure" of the resulting relation will be identical to that yielded by a correctly-formulated query. Example 4.5.7, introduced to illustrate the concept of the Entity Node Chain and its reflection in Reverse Translation text generation was an example of this trap, as applied to two different relations with different predications. In this section the importance of the problem and its solution is elucidated by positing a query environment of extreme simplicity, and demonstrating how the Selection Trap can arise even here. The data to be queried is held in a single, binary relation incorporating a simple predication: the relation PSL, illustrated in Table 2.1. The importance of the Selection Trap can be seen by the following demonstration of four quite simple queries which could be put to this single relation.

7.3.2 Example queries on a single binary relation

A natural language representation of a query.	A sequence of Relational Algebra operations answering the query.
Find...	LIST

1. Persons who speak French.	P1 <- PSL : [L = "French"]
	L1 <- P1 % [P]

2. Persons who speak only French.	P2 <- PSL % [P]
	R1 <- PSL : [L <> "French"]
	R2 <- R1 % [P]
	L2 <- P2 - R2

3. Persons who speak a language other than French.	R1 <- PSL : [L <> "French"]
	L3 <- R1 % [P]

4. Persons who do not speak French.	P2 <- PSL % [P]
	P1 <- PSL : [L = "French"]
	L1 <- P1 % [P]
	L4 <- P2 - L1

7.3.3 Discussion of Example Queries

The first and third queries have relatively straightforward Relational Algebra solutions, inasmuch as the sequence of operators and arguments correlates closely to their equivalent natural language expressions. Even if the final projection -- which, strictly speaking, is required if the derived relation is to match exactly the request -- is omitted, the extra column of information in the resulting binary tuples does not obscure the desired information. But the second and fourth expressions are not at all obvious, and this is particularly the case when seen in the context of the "easy" queries one and three. The natural language expression of the fourth query, so linguistically close to the expression for the first query, invites the unwary user to attempt its solution by making a correspondingly simple modification to the Relational Algebra expression of the first query. Af the first query ("Persons who speak French") was satisfied by the expression "L='French'", then what could be more "natural" than to assume that "Persons who don't speak French" can be found by the expression "L <> 'French'". The result will be a valid query but not the query the user wanted to make. (It will in fact be the solution to Query 3.)

The predisposition to fall into this trap will be all the greater if the user has previously encountered and successfully queried a "many:one" relation of degree 2 (for example, a relation recording persons and their countries of birth), for which the simple substitution of "<>" for "=" <u>will</u> yield semantically valid results. "Persons not born in France" can be thus found but not those who lack French as a language.

All of these queries involve the SELECTION operation. SELECTION is, arguably, the easiest of the relational operators to understand. It is an operation which is implemented on every single-file, micro-computer "database" system. It is interesting to note, then, that the most obvious characteristic of the "difficult" queries, two and four, is that the SELECTION expression involved is exactly the opposite of that suggested in the accompanying natural language expression of the query. To find persons who speak only French, it is necessary to SELECT out those who speak anything else and remove them from the total population. Thus to find the monolingual French-speakers, it is necessary, counter-intuitively, to construct a SELECTION expression where the Language does <u>not</u> equal French. And to find those who don't speak French, it is necessary to first SELECT those for whom "Language = French", and then extract them. Again, the query operation which appears easiest, the SELECTION, has as its most prominent component an expression which appears to be the opposite of its nearest lexical linguistic counterpart in any likely natural language expression. Of course, if we normally expressed ourselves with compounded negations, the previous argument would not hold. If most users tended towards formulating Query 2 in terms like "those persons for whom it is not the case that they speak a non-French Language", then the corresponding sequence of relational operations would exhibit a kind of linguistic/psychological congruency to the natural language expression. This discussion assumes that compounded negations are more difficult for human beings to process, and will therefore be avoided by users in the formulation of their queries. (For a discussion of this question and a brief review of experimental work verifying this assumption, see Johnson-Laird [1983].)

The key operation for the "difficult" queries, two and four, is the final DIFFERENCE operation, the fourth step in the query. Even moderately sophisticated users (the user population assumed by this thesis) are likely by this point to want to confirm their assumptions about the meaning of the relation that would be have been created by this stage.

The Reverse Translation approach to query validation is to give the user another means, in addition to the formal query language, to see what the meaning of his query expression is. The following are the graphs, and Reverse Translations generated from them, associated with each of the queries above.

7.3.4.1 Persons who speak French.

```
P1  <-  PSL  : [L = "French"]
L1  <-  P1   % [P]
```

L1

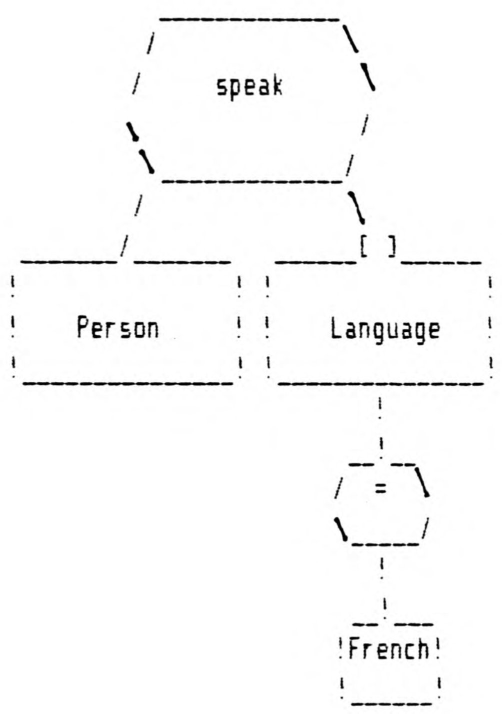


Figure 7.4 The RT graph of L1.

Reverse Translation of L1.

The indicated Person [P], and possibly other Persons,
speaks
at least one Language,
which is French,
and possibly other Languages.

7.3.4.2. Persons who speak only French.

```
P2  <-  PSL  % [P]
R1  <-  PSL  : [L <> "French"]
R2  <-  R1   % [P]
L2  <-  P2   -  R2
```

L2

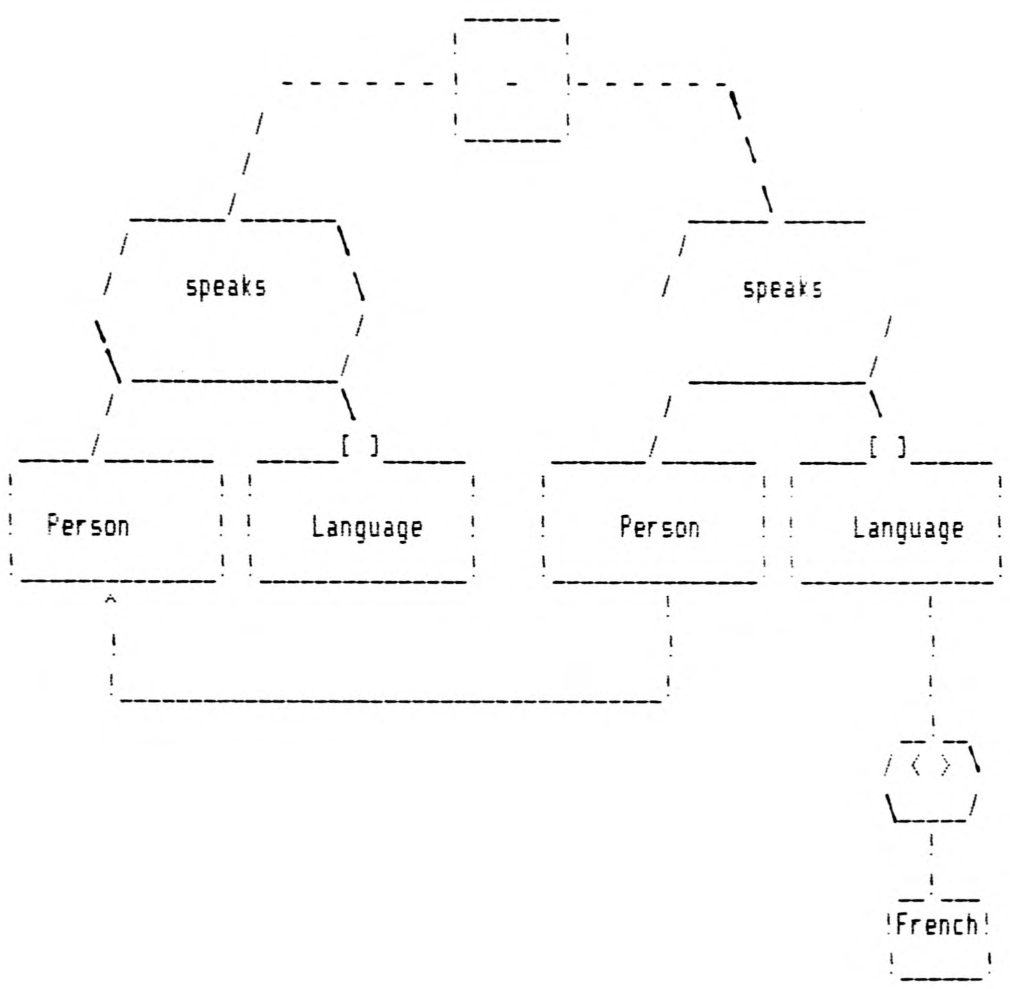


Figure 7.5 The RT graph of L2

Reverse Translation of L2

The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
the indicated Person [P]
speaks
any Language
which is not French

7.3.4.3 Persons who speak a language other than French.

```
R1  <-  PSL  :  [L <> "French"]
L3  <-  R1   %  [P]
```

L3

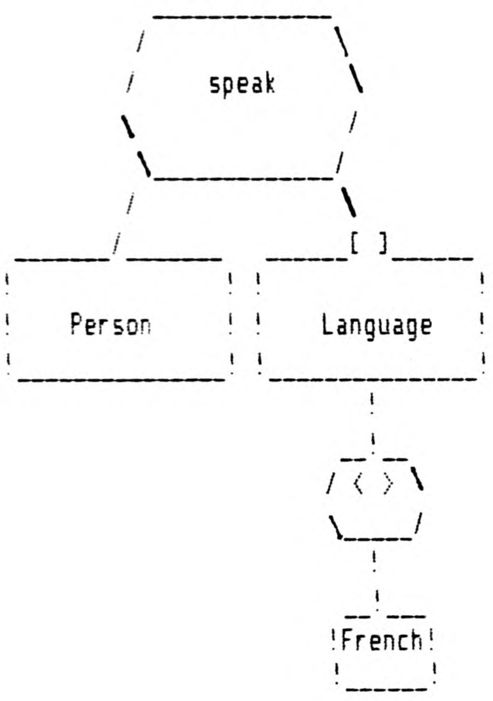


Figure 7.6 The RT graph of L3

Reverse Translation of L3.

The indicated Person [P], and possibly other Persons,
speaks
at least one Language,
which is not French

7.3.4.4. Persons who don't speak French.

```
P2  <-  PSL  % [P]
P1  <-  PSL  : [L = "French"]
L1  <-  P1   % [P]
L4  <-  P2   -  L1
```

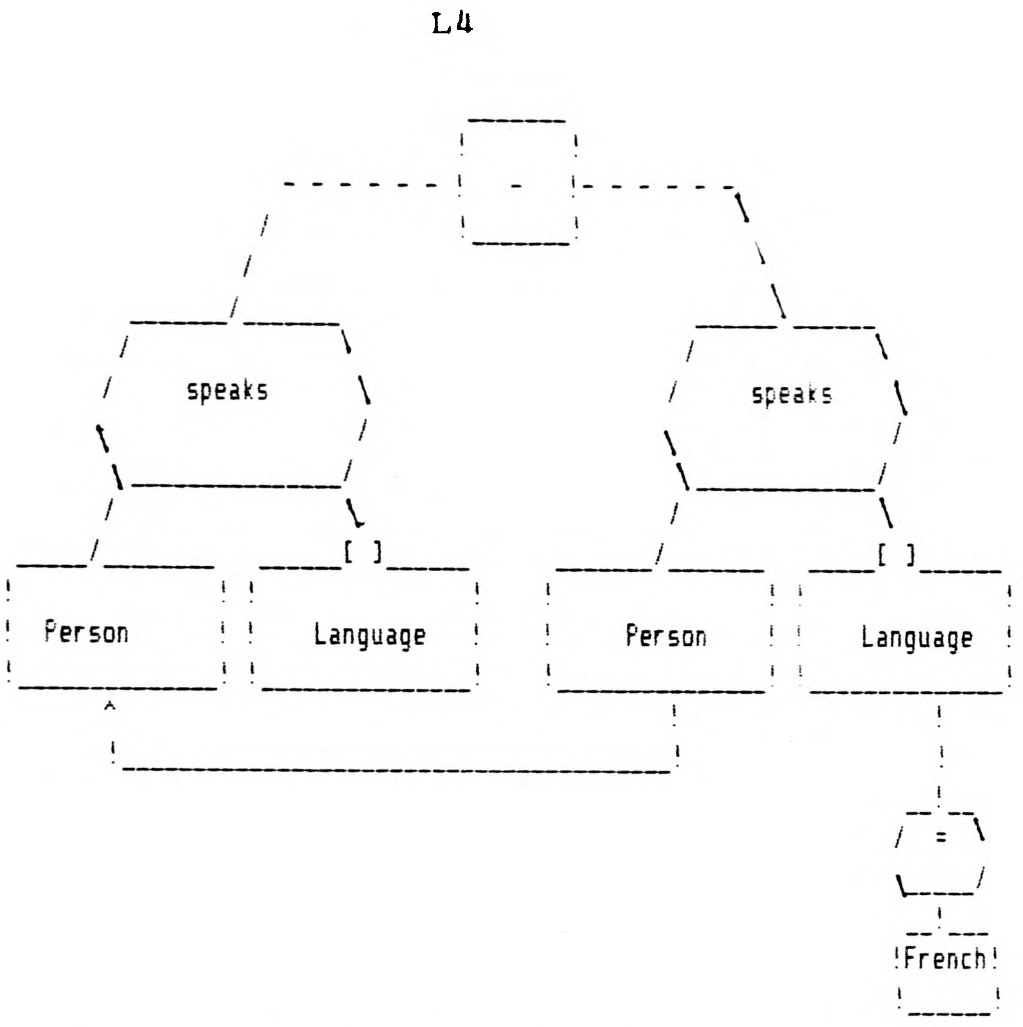


Figure 7.7 The RT graph of L4.

Reverse Translation of L4

```
The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
the indicated Person [P]
speaks
any Language
which is French
```

If the domain of Languages has been marked as "self-identifying" by its creator at domain-creation time, then the Reverse Translation generation algorithm will generate the following, less clumsy, version of the above:

Alternate Reverse Translation of L4

The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
the indicated Person [P]
speaks
French

7.4 The "Difference Trap"

Section 7.3 illustrated the "Selection Trap" for the simplest possible case, a single binary relation. There it was suggested that the counter-intuitive quality of the SELECTION operation needed to formulate certain queries was a possible cause of confusion which Reverse Translation could help to counter. However, an equally plausible argument could be made that the real culprit was the DIFFERENCE operation, involving as it does the the requirement to answer a query by first formulating intermediate relations and then differencing them. Scope for confusion can exist whether or not a SELECTION expression is involved.

This section puts Reverse Translation to a test of its efficacy when DIFFERENCE without SELECTION is involved. The method is applied to "Difference Traps" which can arise when querying a more complex data base, comprising in this example two distinct predications, each with three arguments. The following examples will utilise two relations: Relation P records the promises that suppliers have made to deliver parts to projects. Relation D records those deliveries that were actually made. Both relations have the same surface structure, but different predications. A user wishing to find out about the reliability of Suppliers -- how well promises matched deliveries -- could conceive of a number of queries to put which would yield lists of Suppliers who had demonstrated various modes of unreliability. The following examples demonstrate how Reverse Translation delineates the precise distinction among each of four different ways in which a Supplier can be unreliable.

P

SNO [SUPPLIERS]	PNO [PARTS]	JNO [PROJECTS]
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2
3	1	1
3	1	2
3	2	1
3	2	2
4	1	1
4	1	2
4	2	1
4	2	2
5	1	1
5	1	2
5	2	1
5	2	2

D

SNO [SUPPLIERS]	PNO [PARTS]	JNO [PROJECTS]
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
3	1	1
3	1	2
4	1	1

Table 7.1 Promises and Deliveries

(Note that the tuples of relation D have been aligned with their equivalents in relation P purely as a presentation device to aid the reader in checking the results of differencing D from P. The empty lines in D are not intended to have any other significance.)

Now consider the problem of querying this data base to find out about the performance of suppliers. Who, if any, are completely reliable, who not at all, what are the particular ways in which those suppliers in between the extremes express their reliability and unreliability?

A DIFFERENCE is the simplest meaningful query on this pair of relations which will yield information mentioned in the previous paragraph.

R1 <- P - D

R1

SNO [SUPPLIERS]	PNO [PARTS]	JNO [PROJECTS]
2	2	2
3	2	1
3	2	2
4	1	2
4	2	1
4	2	2
5	1	1
5	1	2
5	2	1
5	2	2

Table 7.2 Unfilled promises

Figure 7.8 illustrates the RT graph yielded by this operation.

R1

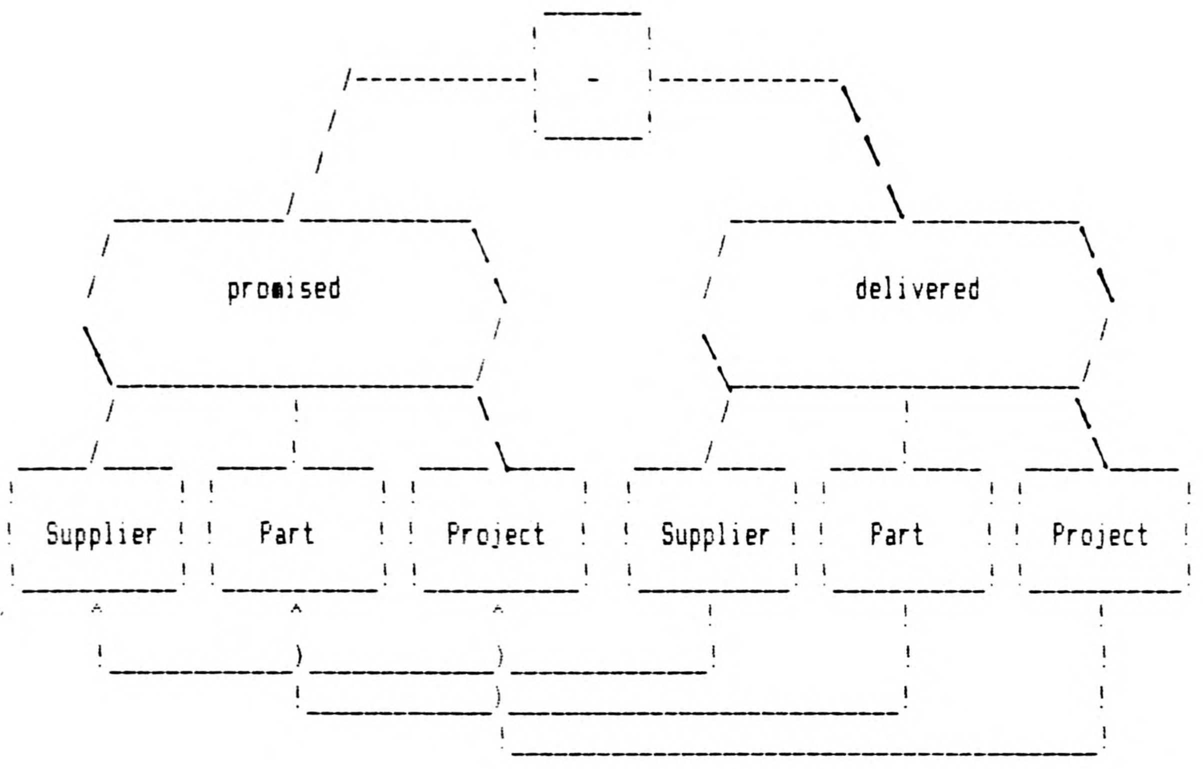


Figure 7.8 <u>The RT Graph of R1</u>

The Reverse Translation of this graph is:

The indicated Supplier [SNO], and possibly other Suppliers,
promised
the indicated Part [PNO], and possibly other Parts,
to
the indicated Project [JNO], and possibly other Projects
but it is not the case that
The indicated Supplier [SNO],
delivered
the indicated Part [PNO],
to
the indicated Project [JNO]

Now let

R2 <- R1 % [SNO]

R2

<u>SNO</u>

[SUPPLIERS]
2
3
4
5

Table 7.3 Unreliable Suppliers I

The RT graph of R1 will now be modified as illustrated in Figure 7.9.

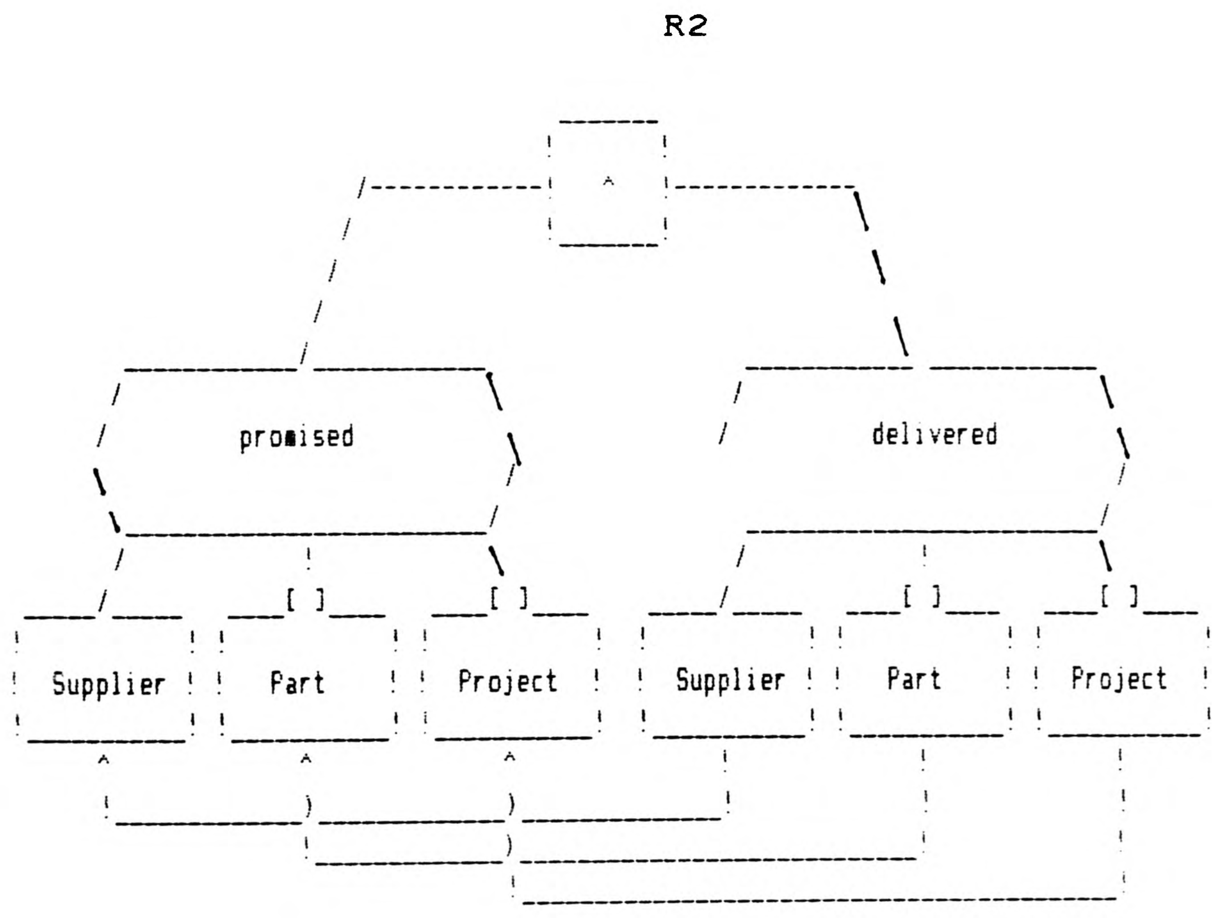


Figure 7.9 The RT Graph of R2

The Reverse Translation of this graph is:

```
The indicated Supplier [SNO], and possibly others,
promised
at least one Part, and possibly others,
to
at least one Project, and possibly others
but it is not the case that
The indicated Supplier [SNO],
delivered
that Part,
to
that Project
```

It is useful to contrast the RT generated from the above operations with that generated by a superficially-similar query.

```
S3  <-  P  %[SNO]
S4  <-  D  %[SNO]
R3  <-  S3  -  S4
```

S3	S4	R3
SNO	SNO	SNO
[SUPPLIERS]	[SUPPLIERS]	[SUPPLIERS]
1	1	
2	2	
3	3	
4	4	
5		5

Table 7.4 Unreliable Suppliers II (in R3)

The RT graph which would be created as a result of the above query is illustrated in Figure 7.10.

R3

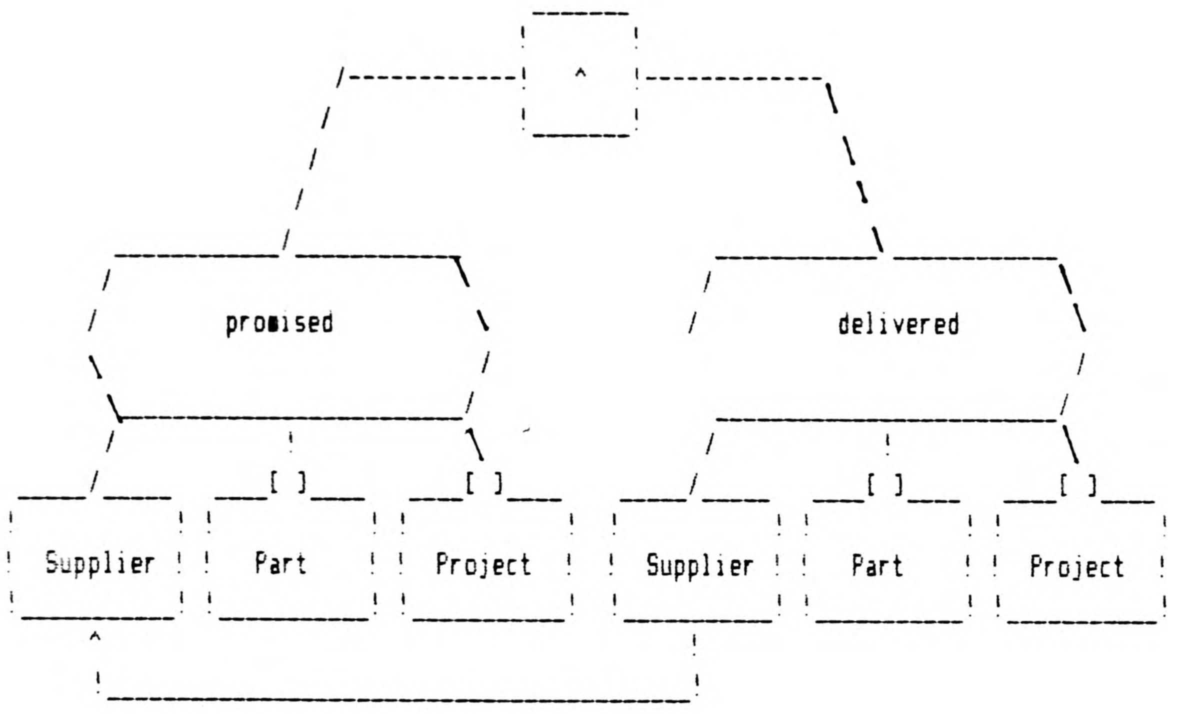


Figure 7.10 <u>The RT Graph of R3</u>

The Reverse Translation of this graph is:

```
The indicated Supplier [SNO], and possibly other Suppliers,
promised
at least one Part, and possibly other Parts,
to
at least one Project, and possibly other Projects
but it is not the case that
The indicated Supplier [SNO],
delivered
any Part,
to
any Project
```

The points to note here are:

(1) the second occurence of Part and Project do not refer back to the first occurences of these entities.

(2) the use of the phrase "any" when quantifying the projected-out entities of the right-hand graph. (The use of the phrase "at least one" instead of "any" would definitely convey the wrong impression.)

Two other queries to find sometimes unreliable suppliers are:

```
   S4  <-  P %[SNO, PNO]
   S5  <-  D %[SNO, PNO]
   R5  <-  S4  -  S5
```

S4		S5		R5	
SNO	PNO	SNO	PNO	SNO	PNO
[SUPPLIERS]	[PARTS]	[SUPPLIERS]	[PARTS]	[SUPPLIERS]	[PARTS]
1	1	1	1		
1	2	1	2		
2	1	2	1		
2	2	2	2		
3	1	3	1		
3	2			3	2
4	1	4	1		
4	2			4	2
5	1			5	1
5	2			5	2

Table 7.5 <u>Unreliable Suppliers (III) and Parts</u>

Extracting the Supplier Numbers of delinquent suppliers through a projection,

```
      R6  <-  R5  %[SNO]
```

R6
SNO
[SUPPLIERS]
3
4
5

Table 7.6 <u>Unreliable Suppliers III</u>

R6

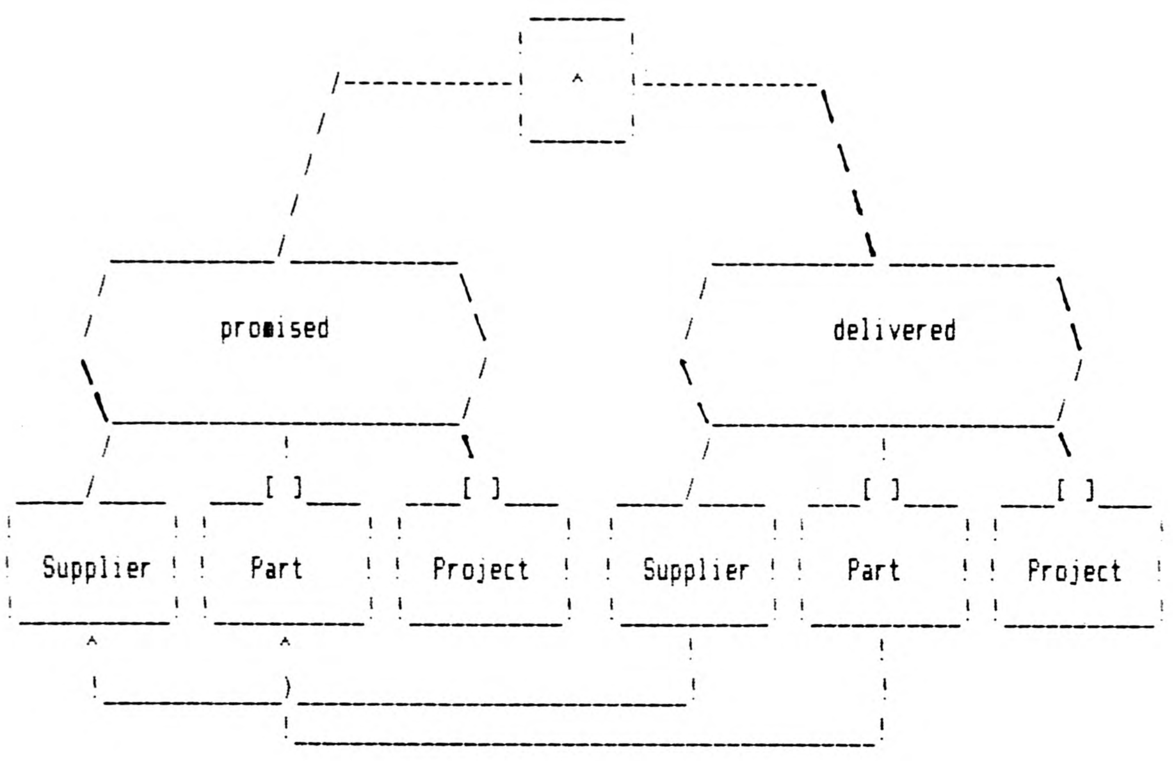


Figure 7.11 <u>The RT Graph of R6</u>

The Reverse Translation generated by this graph is:

```
The indicated Supplier [SNO], and possibly other Suppliers,
promised
at least one Part,
to
at least one Project,
but it is not the case that
The indicated Supplier [SNO],
delivered
that Part,
to
any Project
```

Another kind of defaulter would be found with the following expression.

```
S6  <-  P   % [SNO, JNO]
S7  <-  D   % [SNO, JNO]
R7  <-  S6  -  S7
```

S6

SNO	JNO
[SUPPLIERS]	[PROJECTS]
1	1
1	2
2	1
2	2
3	1
3	2
4	1
4	2
5	1
5	2

S7

SNO	JNO
[SUPPLIERS]	[PROJECTS]
1	1
1	2
2	1
2	2
3	1
3	2
4	1

R7

SNO	JNO
[SUPPLIERS]	[PROJECTS]
4	2
5	1
5	2

Table 7.7 <u>Unreliable Suppliers IV (R7) and Projects</u>

Extracting the supplier numbers of delinquent suppliers through a projection,

```
        R8  <-  R7  %[SNO]
```

we get

R8

SNO
[SUPPLIERS]
4
5

Table 7.8 <u>Unreliable Suppliers IV</u>

The corresponding graph is displayed in Figure 7.12.

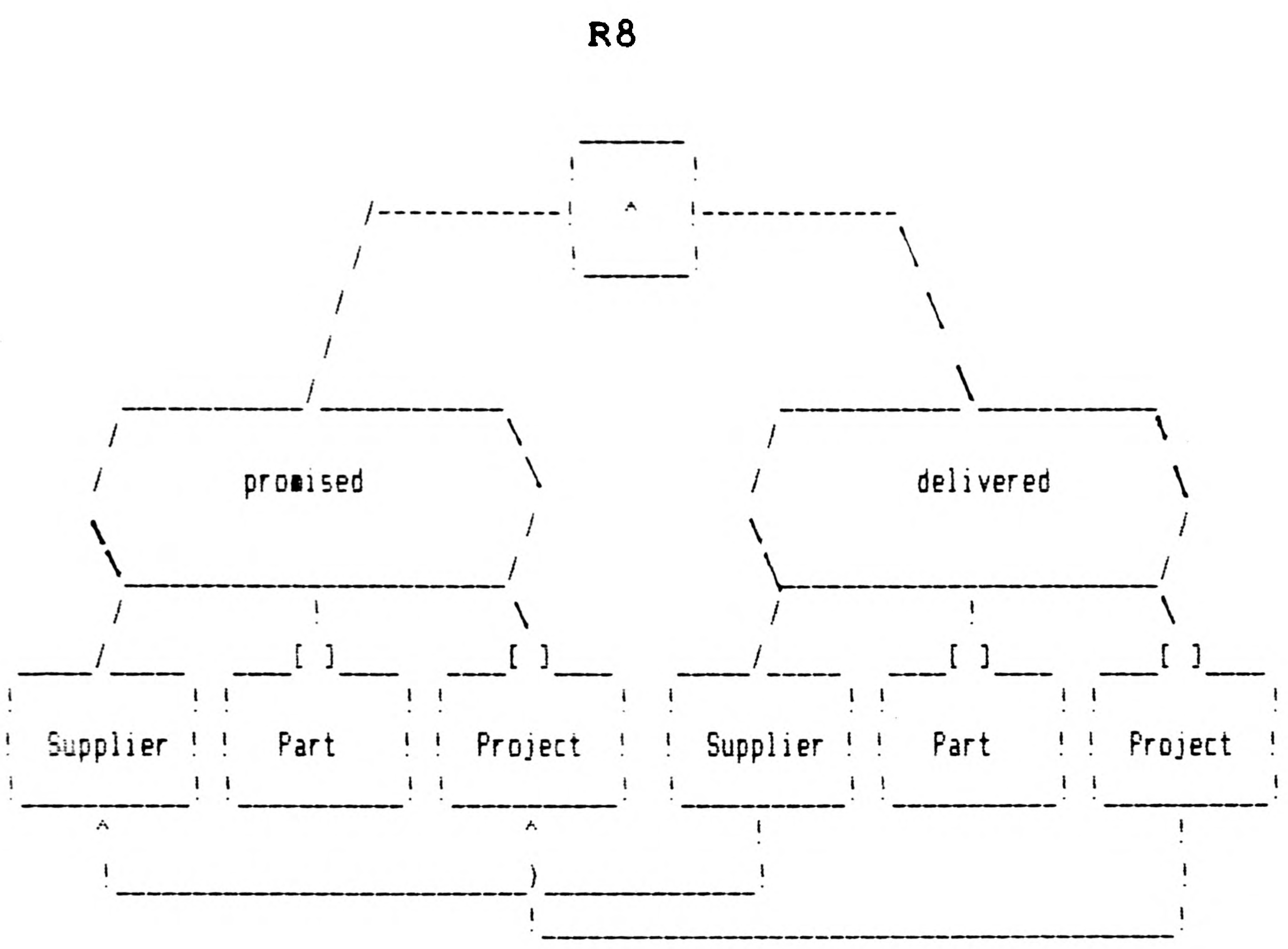


Figure 7.12 The RT Graph of R8

The Reverse Translation generated by this graph is:

```
The indicated Supplier [SNO], and possibly other Suppliers,
promised
at least one Part,
to
at least one Project,
but it is not the case that
The indicated Supplier [SNO],
delivered
any  Part,
to
that Project
```

CHAPTER EIGHT

CONCLUSIONS

8.1 Introduction

The data structure and associated algorithms described in the preceding chapters generate useful Reverse Translations for the kinds of relational algebra expressions which have been used to illustrate the technique. These expressions include non-trivial queries, for which clarification might plausibly be argued to be helpful to a non-sophisticated user. However, the Reverse Translation method as so far developed has limitations when it confronts certain classes of query expression. In the final chapter some of these limitations are considered and possible solutions to the problems they presentare outlined. Directions for further research are indicated and an overall assessment of the practical usability of Reverse Translation is made.

8.2 Methods for simplifying utterances

8.2.1 Predication Lowering

The present Reverse Translation system gives all Predication Nodes "equal status" as nodes in a Reverse Translation tree. Joins and set operations will compose Predication Nodes via Logical Nodes of which they will be the immediate descendants. No Predication Node can be a descendant of an Entity Node. This has the consequence that certain plausible sequences of operations will yield unwieldy Reverse Translations.

Consider a personnel database with several binarypredications, relating PERSONS located-in CITIES, speaking LANGUAGES, qualified-in PROGRAMMING-LANGUAGES, and assigned-to PROJECTS. Assume that Edegree information ("and possibly others") is suppressed and that all domains except PROJECTS have been declared self-identifying. Even then, a series of selections, joins and projections to extract everyone who is located in London, speaks French, is qualified in COBOL and assigned to Project P1 will yield a Reverse Translation tree generating the following Reverse Translation: "The indicated PERSON [P] is located in London, AND the indicated Person [P] speaks French, AND the indicated PERSON [P] is qualified in Cobol AND the indicated PERSON [P] is assigned to a PROJECT whose Project Number equals P1." (Of course the exact order in which each predication is uttered will depend on the order in which the relation embodying it is joined to the final derived relation.)

This is arguably a clumsy construction, and could be improved if we had a rule like the following: when two predication nodes are descendants of an AND node, and are linked by one, and only one Entity Node Chain,and if this graph is then a participant in the creation of a new AND-linked graph, first do the following to it: transfer the right-hand Predication Node (and its descendants) to the Principal Entity Node using the Qualification Sub-Graph transferral algorithm. Mark the (now disconnected) Entity Node in the transferred sub-graph with a "dummy" marker so that the RT generation algorithm will utter its relative pronoun when it is encountered during text generation. Carrying out these transformations on the hypothetical graph posited above at each operation would yield a graph giving the following Reverse Translation: "The indicated PERSON [P] who is located in London AND who speaks French AND who is qualified in COBOL is assigned to a PROJECT whose Project Number equals P1..." Figure 8.1 demonstrates the effect of this transformation.

8.2.2 Entity Sub-set naming

Another method for preventing a Reverse Translation from becoming unintelligbly complex, would be to allow the user to intervene at points in the process of query composition to "trim" entity nodes and rename them. This intervention would be equivalent to the user's re-defining a dervied relation as a Base Relation, using a set of

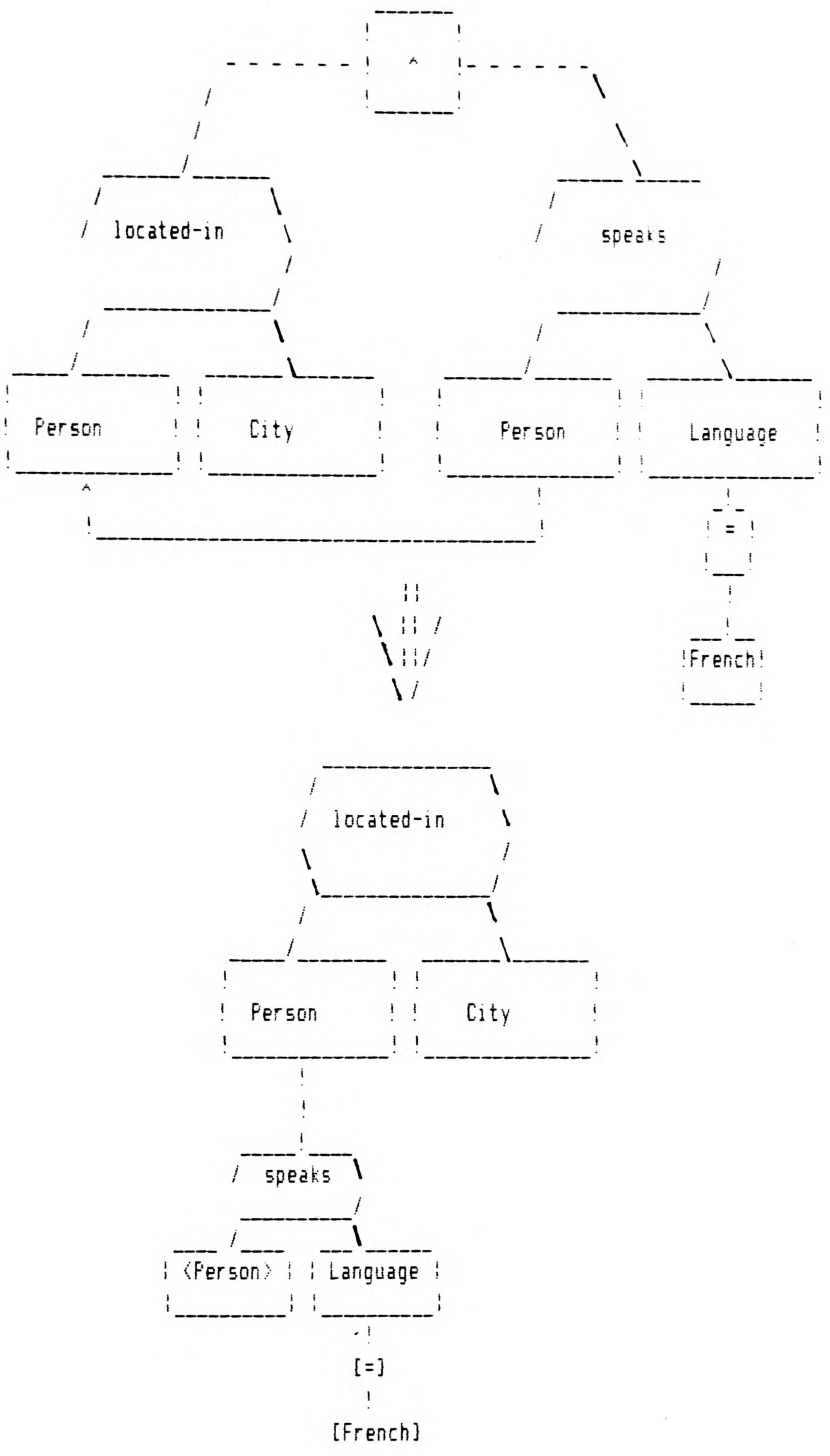


Figure 8.1 __Predication-lowering__

Predication phrases or Entity names meaningful to him and bearing adequate semantic information to distinguish the derived relation from any other. This would be done once the user is satisfied that the derived relation is the desired one. The information carried in the complex graphs of the particular derived relation would be transferred to the predication phrase set alone, and the relation would "start over" with a minimal graph. This may be analogised to the way in which new nouns and adjectives are evolved in natural language through a process of spontaneous creation, to replace involved subordinate descriptive clauses.

The example Reverse Translation cited in the previous section might, under this approach, be re-defined by a user to something like the following: "The indicated French-speaking London Cobol-programmer is assigned to a Project whose Project number is P1."

Figure 8.2 demonstrates the effect of such a transformation on an RT graph.

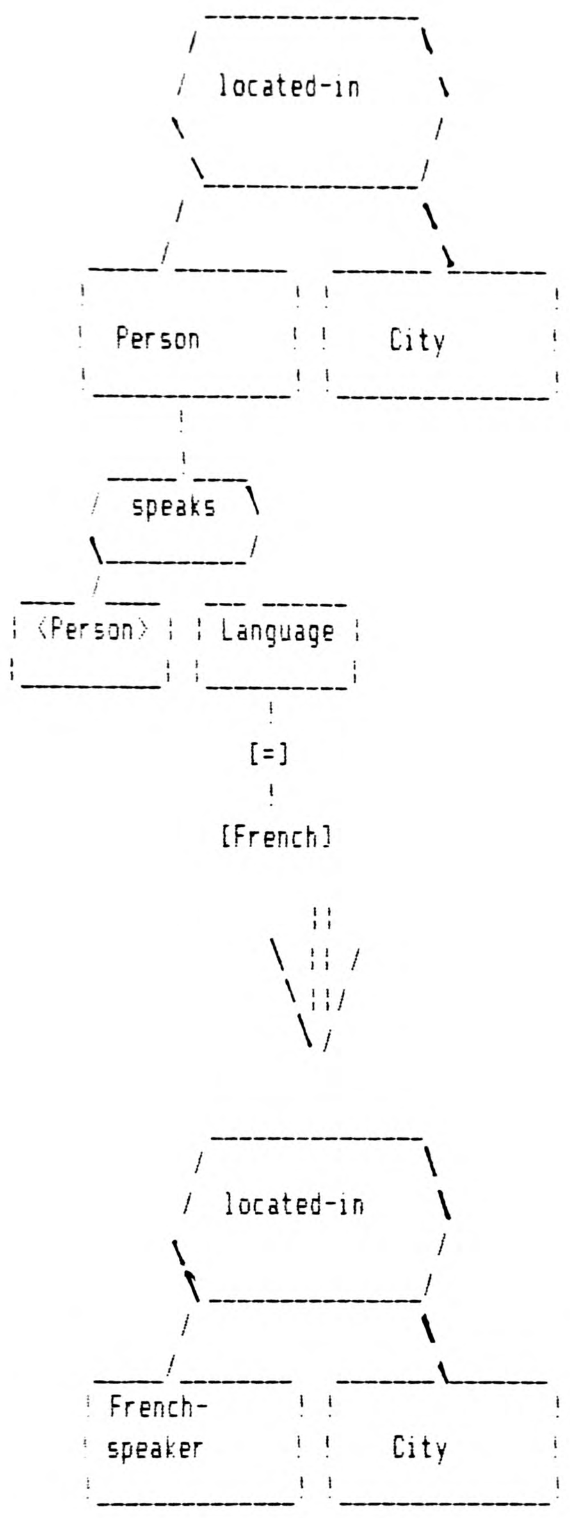


Figure 8.2 Turning a Qualified Entity into an Entity Sub-Set

8.3 Indicating Scope

The system as currently implemented does not provide any indication of the scope of logical connectives. This may be seen by considering a query placed on the relations introduced in Chapter 7, PSL and PRL. Suppose a user wants to compose a query expression to find those who speak at least one language, but do not read any which they speak. (They may read some, but none which they speak.) A sequence of expressions which will yield the desired answer is

```
R1 <- PSL ^ PRL
R2 <- R1  % [P]
R3 <- PSL % [P]
R4 <- R3  - R2
```

The RT graph resulting from this sequence is shown in Figure 8.3.

R4

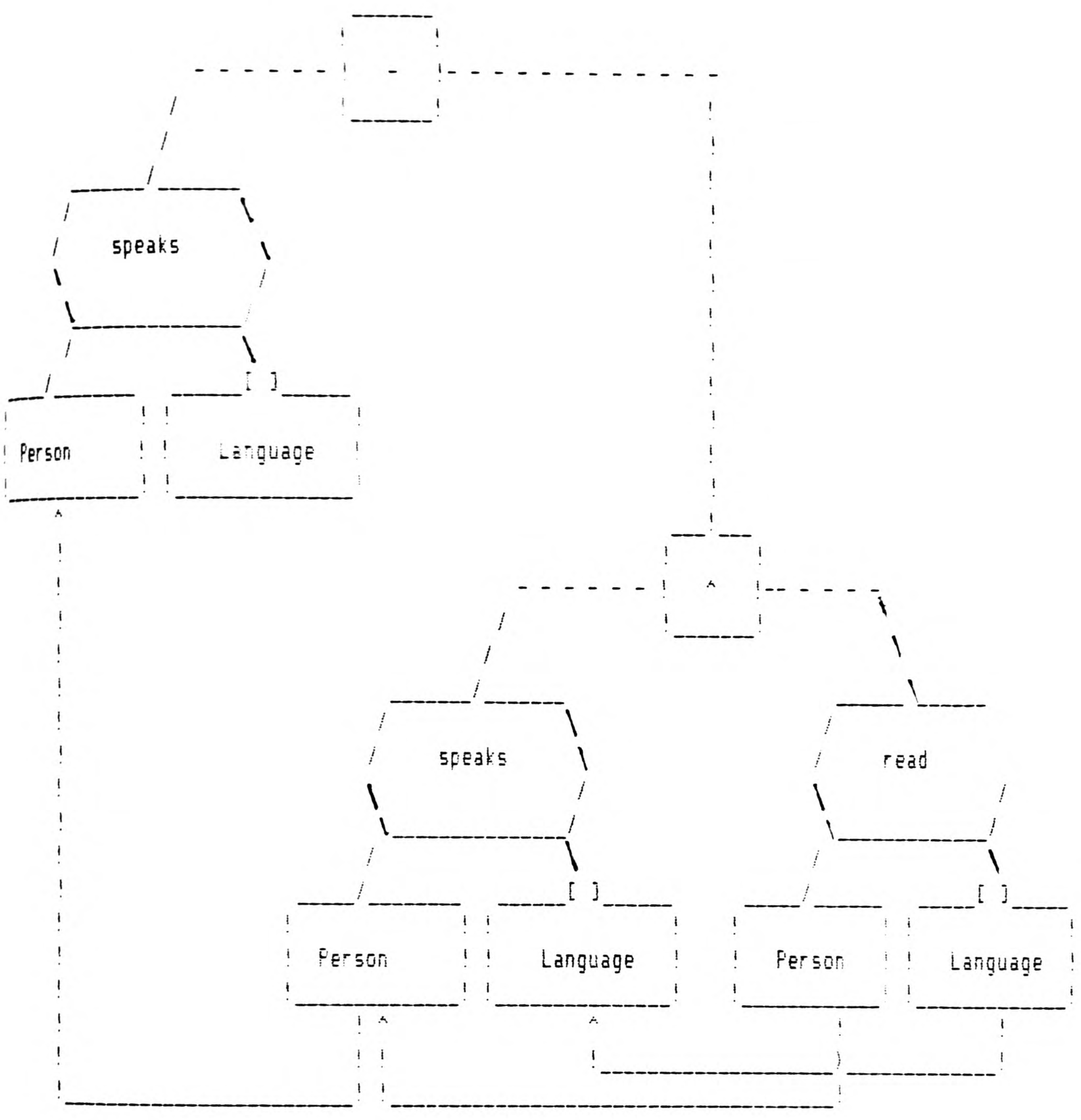


Figure 8.3 The RT graph of R4.

Reverse Translation of R4

The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
the indicated Person [P]
speaks
at least one Language
and
the indicated Person [P]
reads
that Language

The Reverse Translation as illustrated is ambiguous. There are two possible readings, which are illustrated by indenting to indicate the scope of the NOT Logical Node.

Interpretation I:

```
The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
     the indicated Person [P]
     speaks
     at least one Language
and
     the indicated Person [P]
     reads
     that Language
```

Interpretation II:

```
The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
     the indicated Person [P]
     speaks
     at least one Language
     and
          the indicated Person [P]
          reads
          that Language
```

The second interpretation is the correct one. (Interpretation I is self-contradictory.) Adding a facility to the current system for some sort of automatic indenting of Reverse Translation text would not be difficult. However, its actual effect on aiding correct user perception of connector scope may not be clearcut and would require empirical verification. (Other methods of graphically indicating scope, such as enclosing logically-connected blocks within brackets, using a distinct typeface, or setting them apart with blank lines, or some combination of these, might be equally efficacious or superiour.) The pragmatics of aiding user perception are beyond the scope of this thesis, but obviously are a fruitful field for research.

8.4 Handling Double Negations

It can occur that a query will yield a graph in which the descendant of a NOT Logical Node is itself a NOT Logical Node. (This happens, for example, in certain queries with universal quantifiers.) The speakers and readers example from the previous section can provide us with an example of this problem, too. Consider the query sequence yielding those who speak at least one language, and who read all which they speak.

R1 <- PSL - PRL	(Person-language pairs where the person speaks a language he does not read.)
R2 <- R1 % [P]	(Persons from the above set.)
R3 <- PSL % [P]	(All persons who speak a language.)
R4 <- R3 - R2	(Persons who speak a language, purged of those who speak one they do not read.)

This sequence will yield the graph in Figure 8.4:

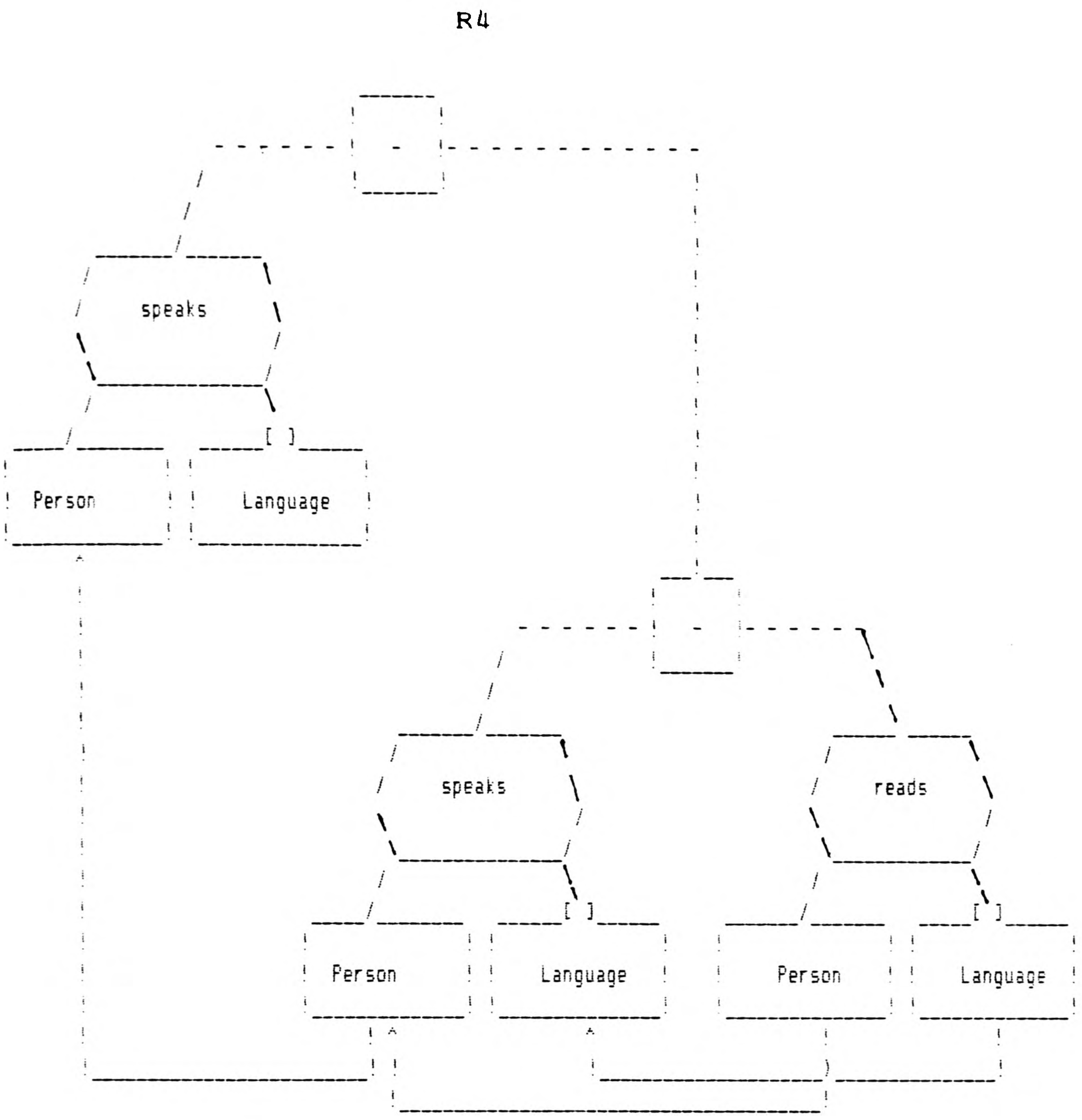


Figure 8.4 <u>The RT graph of R4</u>

The Reverse Translation of R4, indented to show the scope of the Logical Nodes, is

```
The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
     the indicated Person [P]
     speaks
     at least one Language
     but it is not the case that
          the indicated Person [P]
          reads
          that Language
```

The Predication Node defining procedure solicited from the user both positive and negative senses of the Main Predication Phrase. @ere is where the latter could be used. Whenever the Reverse Translation algorithm encounters a double negation situation, as defined above, it could alter the phraseology it generates for the second NOT Logical Node from the standard "but it is not the case that" to simply "but", and then utter the negative sense of the descendant predication node. Such a strategy applied to the graph in Figure 8.3 would yield the following Reverse Translation:

```
The indicated Person [P], and possibly other Persons,
speaks
at least one Language
but it is not the case that
     the indicated Person [P]
     speaks
     at least one Language
     but
          the indicated Person [P]
          does not read
          that Language
```

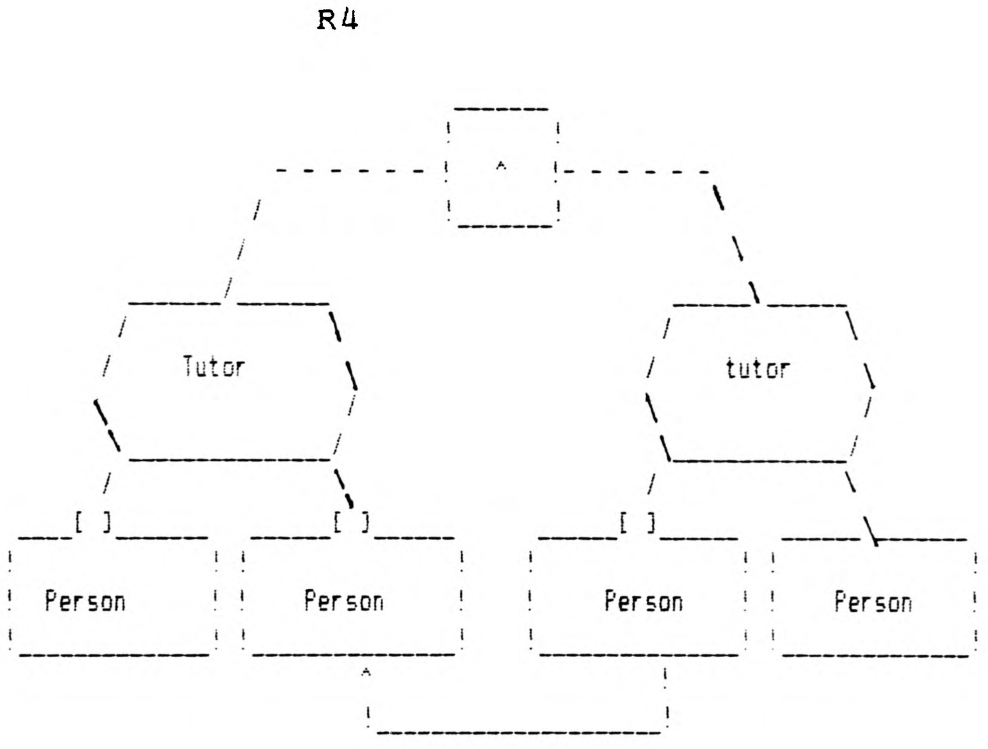
The second translation (combined with some method of indicating scope) is perhaps marginally clearer than the first, although the improvement is not overwhelming.

8.5 <u>Ambiguous reference</u>

The potential of multiple reference arise, consider a binary relation T, with attributes Tutor and Student, recording who tutors whom. Both attributes are drawn from the domain PERSONS. If we wish to find all those who are tutored by someone who is himself a tutee, we must, after suitable renaming of attributes, join this relation to itself over the tutee attribute in one relation and the tutor attribute in the other.

```
R1 <- @ Tutors [ Student -> Both ]
R2 <- @ Tutors [ Tutor   -> Both ]
R3 <- R1 * R2
R4 <- R3 % [ Student ]
```

A graph of R4 is



The Reverse Translation generated from the above graph is

```
At least one Person
tutors
at least one Person
and
that Person
tutors
the indicated Person [P] and possibly other Persons
```

This is unsatisfactory because it is not clear to whom the "that" refers. A possible solution to this problem, albeit a clumsy one, is to print Node-id values after each domain name. This would be going some distance from the "spirit" of Reverse Translation. It could be argued that if it were necessary to generate such "unnatural" natural language, then the cause of user understanding would be just as well served by a display of the Reverse Translation graph as well. As in the discussion of connective scope, the actual solution chosen should be a pragmatic question settled by experimentation.

Another possible solution would be to modify the utterance algorithm to recognise such ambiguous situations by scanning the stack of uttered Entity Nodes as it is built up, and generating additional clarifying text when appropriate (for instance, "the second one" after the ambiguous "that person" in the example above).

8.6 Universal Quantification

It is possible to construct queries which require a complicated sequence of operations resulting in a correspondingly complex graph which will host a correspondingly baroque Reverse Translation, but for which a relatively simple meaning can be assigned. A good example of this would be queries involving universal quantification, obtained via a Cartesian product and a repeated difference. The Reverse Translation of such a query, although accurate, is of little value in validating the query sequence.

Consider just the relation PRL, relating PERSONs and the LANGUAGEs they read. In the absence of an explict universal quantification operator, the following sequence of operations must be performed to find those who can read *all* languages.

```
R1 <- PRL % [P] (those who have read  at least one
                 language, and, )

R2 <- PRL % [L] (the  languages  they have  read)

R3 <- R1 * R2   ( all possible reader-language
                  pairs, existing and non-existant --
                  the Cartesian product of R1 and R2)

R4 <- R3 - PRL  (remove   from    R3    all    actual

                 reader-language pairs -- a person
                 who has read all  languages will
                 thus  have  every  instance of  his
                 name removed from R4)

R5 <- R4 % [P]  (leaving  those who  have  not
                 read all languages)

R6 <- R1 - R5   (remove these from the general student
                 population to  get  persons who have
                 read all languages)
```

This sequence results in the graph in Figure 8.5:

R6

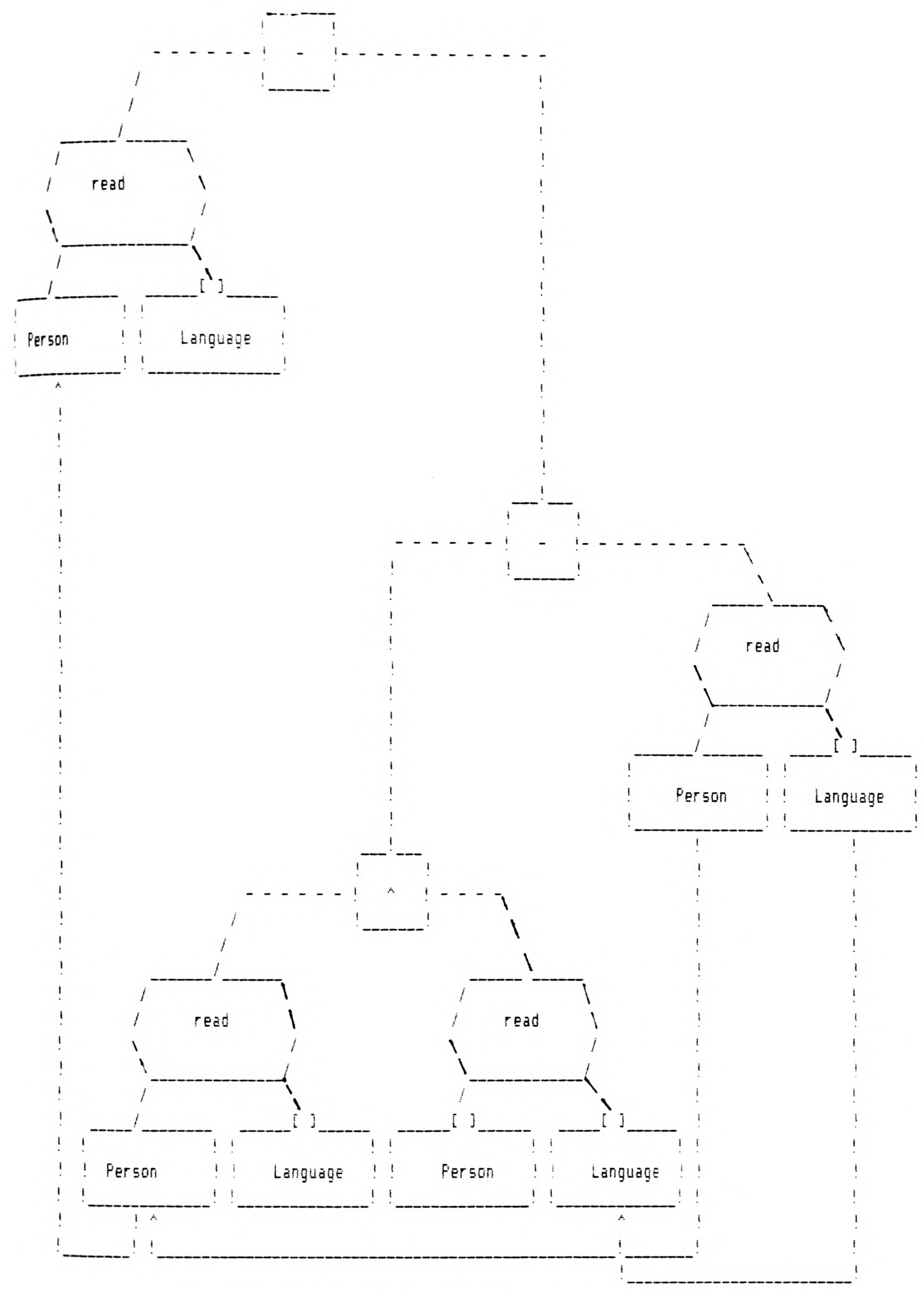


Figure 8.5 The Graph of R6 -- polyglot readers

The Reverse Translation, indented to show scope, that is generated from this graph is:

```
The indicated Person [P]
reads
at least one Language
but it is not the case that
     the indicated Person [P]
     reads at least one Language
     and
          at least one Person
          reads
          at least one Language
     but it is not the case that
          the indicated Person [P]
          reads
          that Language
```

It does not require field-testing to assert that this is not helpful. What is wanted is a Reverse Translation along the lines of "The indicated Person [P] reads all Languages". Of course, provision of an explicit universal quantification operator in the query language available to the user would allow him to avoid the necessity of creating such complex sequences as the example. But the point is, there is nothing to prevent a naive user from constructing the above query and we would, ideally, like to be able to handle it elegantly. An "intelligent" Reverse Translation system could recognise such sequences as the above as and generate suitably terse but meaningful text.

One approach to a solution to this problem might be to attempt to devise a special "fix", possibly involving pattern recognition techniques, which would specifically trap and handle instances of the above graph. But universal quantification can occur in indefinitely many other queries as well, including some which can appear deceptively simple. To see a striking example of this, we may follow through the steps necessary to answer Query number 27 in Chapter 7 of [Date, 1981].

"Get JNO values for projects which obtain at least some of every part they use from supplier S1."

This query assumes the existence of the following relation: (Sample data values for the base relation and each intermediate relation are also provide.)

```
SPJ
SNO       PNO  JNO
[SUPPLIERS] [PARTS] [PROJECTS]
```

SNO	PNO	JNO
S1	P1	J1
S1	P1	J2
S1	P1	J3
S1	P2	J1
S1	P2	J2
S1	P3	J2
S2	P1	J2
S2	P1	J4
S2	P2	J4
S3	P2	J2
S3	P2	J4
S3	P3	J4
S4	P3	J2
S4	P3	J3
S4	P4	J3

Table 8.1 Suppliers, parts, projects

```
T1 <- SPJ : [ SNO = 'S1' ]
```

SNO	PNO	JNO
S1	P1	J1
S1	P1	J2
S1	P1	J3
S1	P2	J1
S1	P2	J2
S1	P3	J2

T2 <- T1%[PNO,JNO]

PNO	JNO
P1	J1
P1	J2
P1	J3
P2	J1
P2	J2
P3	J2

T3 <-SPJ%[PNO,JNO]

PNO	JNO
P1	J1
P1	J2
P1	J3
P1	J4
P2	J1
P2	J2
P2	J4
P3	J2
P3	J3
P3	J4
P4	J3

T4 <- T3 - T2

PNO	JNO
P1	J4
P2	J4
P3	J3
P3	J4
P4	J3

T5 <- T4 %[JNO]

JNO
J3
J4

T6 <- SPJ %[JNO]

JNO
J1
J2
J3
J4

T7 <- T6 - T5

JNO
J1
J2

Table 8.2 Intermediate relations

T7

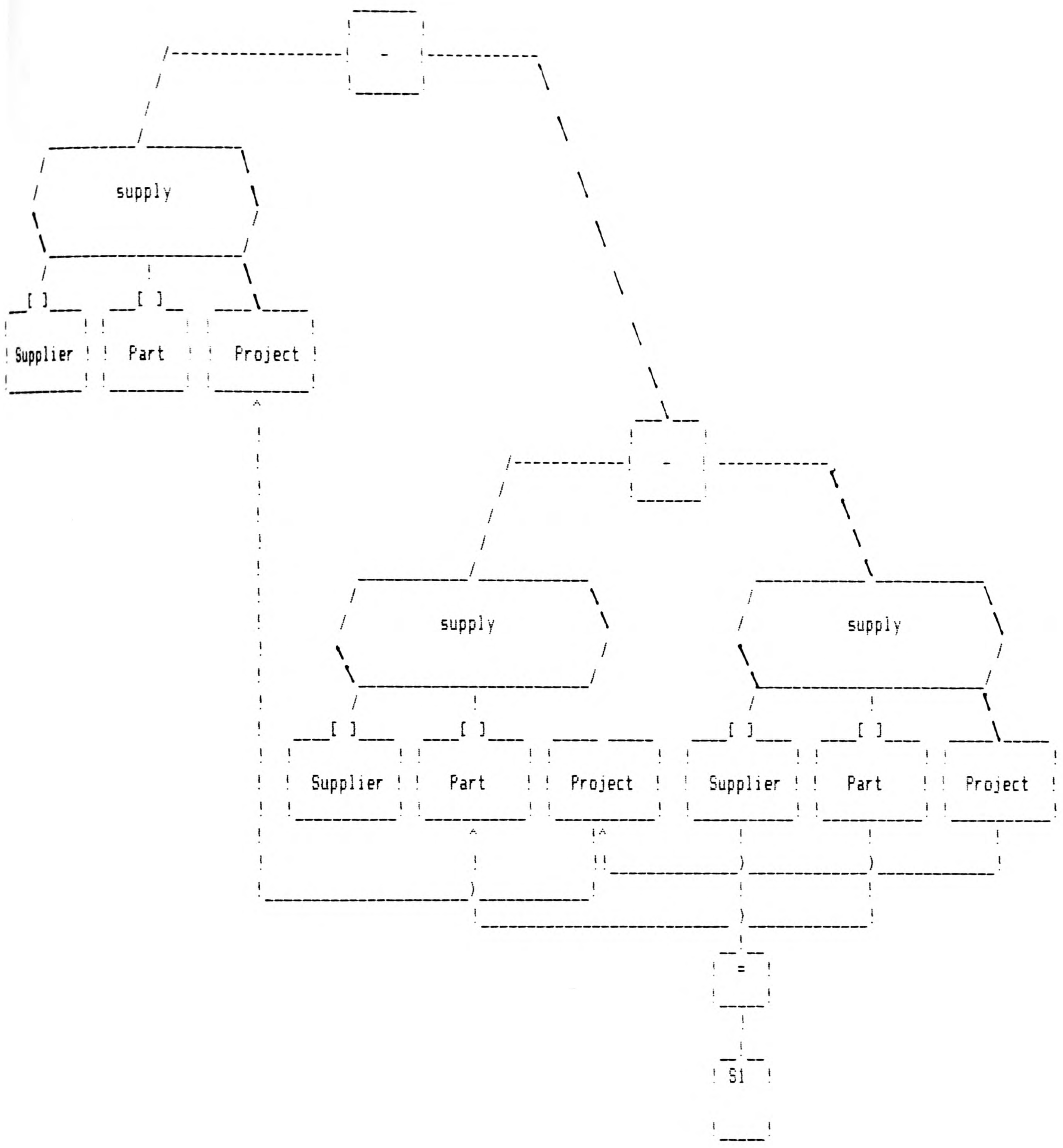


Figure 8.6 RT Graph of T7

The Reverse Translation (indented to indicate scope) which this graph would generate is:

```
At least one Supplier
supplies
at least one Part
to
the indicated Project [PNO], and possibly other Projects
but it is not the case that
   at least one Supplier
   supplies
   at least one Part
   to the indicated Project [PNO]
   but it is not the case that
      at least one Supplier
      whose Supplier-Number = S1
      supplies
      that Part
      to the indicated Project
```

As in the case of the straightforward universal quantification query illustrated in the previous section, this is not helpful. There is too great disparity, in terms of the mental effort needed to equate them, between the "description" of T7's semantics generated by the Reverse Translation technique, and our common sense rendering of this relation as "projects which get at least some of every part they use from Supplier S1". In the next section this problem is taken up more generally and its implications discussed.

8.7 An Assessment of Reverse Translation and Directions for Further Research

The Reverse Translation system presented in this thesis shares the limitations of all Natural Language processing systems which work by syntactic rules alone. Natural Language is a medium which pre-supposes a rich corpus of shared knowledge on the part of its direct or indirect originator and its receiver. The classic illustration of this truism is the problem of algorithmically determining the reference for the pronoun "he". Consider the sentence, "The boy ran towards the river, and then he fell". A language processing algorithm can easily be imagined which could correctly substitute "the boy" for "he" in a system-generated paraphrase of this sentence. A rule like "substitute the most recently-generated animate noun" would suffice. Now consider the sentence, "The gunman fired at the victim, and then he calmly reloaded." Whereas the human information processor would have no difficulty in substituting "the gunman" for "he", no algorithm based purely on syntax and limited properties of words could decide whether it was the gunman or his victim who reloaded. Knowledge about the world of gunmen, guns, firing and victims is necessary to handle this sentence, which is superficially similar to the first example.

Turning to the problem of simplifying Reverse Translations, we may consider the example shown in Chapter Seven, Section 7.3.4.2. The Reverse Translation describes a person for whom "it is not the case that the indicated Person speaks any language which is not French." A moment's reflection yields the much tighter and easier to understand paraphrase "speaks <u>only</u> French". As suggested in the preceding section, it is possible that further purely structural graph manipulations could be added to the Reverse Translation system, this time to deal with this class of query. For instance, a close equivalent to the quoted paraphrase would be generated by a rule like, "for a projected-out Entity Node which is a descendant of a negated Predication Node and which is not part of an Entity Node Chain, invert all Logical and Comparison Nodes in its Qualification Sub-graph and set its Edegree to one."

However, an ever-more-elaborate set of rules to cover multiple special cases is an unsatisfying method of approach to this problem. One possible approach to a solution is some sort of deductive component in the system, with a relatively small set of powerful, general rules of the type, "if some X has a relationship with a set Y, but does not have a relationship with a sub-set of Y, then it has an exclusive relationship with the complement of that sub-set." It may be conjectured that rules of this type, perhaps embodied as mental models, are what allow humans to deduce that "if someone speaks a language, but he doesn't speak any language that isn't French, then he speaks French (and it alone)." Further elaboration of such a system is beyond the scope of this thesis, but should be noted as a potentially fruitful research direction.

8.7 Practical Applications of Reverse Translation

For the reasons delineated in the previous section it is unlikely that any Reverse Translation system based on purely syntactic manipulations could ever serve as an unmediated and general-purpose validation system for naive users placing unrestrictedly-complex queries on databases. However, a useful application of a robust Reverse Translation, elaborated in the directions indicated in this chapter, might be found in the area of user training. Currently, a user who wishes to learn the semantics of a formal query language without expert human assistance has two learning techniques available. He can follow examples provided in written or on-line tutorial exercises, if these are available. In this case he may find that the examples provided are taken from an unfamiliar application domain, or do not illustrate in sufficient detail the kind of query sequences which are of interest to him. A second learning approach would be for the user to create and populate his own database schema and then apply simple query operations to it, studying the results and generalising from them. This has the disadvantage that false generalisations can be drawn, taking what may only be true contingently -- due to the particular data values present in a relation -- for a necessary truth.

A Reverse Translation tutorial session would allow the user to create his own database schema, including the predication strings linking the domains present in each relation. He could then see the effect, illustrated by Reverse Translations, of increasingly-complex query expressions. Of course, at some point, given the limitations of purely syntax-based natural language generation, he would be confronted with Reverse Translations whose complexity rendered them of little pedagogic utility. If, however, the semantics of each query language operation had been illustrated and taught before this point was reached, then Reverse Translation would have proved to be a useful learning device. Whether or not this scenario is practical is itself a matter for further research.

BIBLIOGRAPHY

Bibliography

Robert B. Anderson
Proving Programs Correct
John Wiley and Sons, 1979

Yigal Arens, David Chin and Robert Wilensky
Talking to UNIX in English: an Overview of UC
Communications of the ACM XXVII(6):574-591, June 1984

Eugene Ball, Phil Hayes and Raj Reddy
Computers with Natural Language Communication Skills
Computer Science Research Review 1979-80:39-51
Carnegie-Mellon University

R. Bell and P. Gray
Use of Simulators to Help the Inexpert in Automatic Program Generation
IFIP 1979: 613-628

Boris Beizer
Software Testing Techniques
Van Nostrand Reinhold, New York, 1983

K. Biss, R. Chien and F. Stahl
R2 -- a Natural Language Question-Answering System
AFIPS Conference Proceedings, Spring Joint Computer Conference XXXVIII:303-308, 1971

Robert W. Blanning
Conversing with Mangement Information Systems in Natural Language
Communications of the ACM XXVII(3):201-207, 1984

Daniel G. Bobrow
Natural Language Interaction Systems
Picture Language Machines:31-65
Academic Press, 1970

Leonard Bolc
Natural Language Question Answering Systems
Carl Hanser Verlag 1980

Jeffrey Bonar, Elliot Soloway and Kate Ehrlich
Cognitive Strategies and Looping Constructs: an Empirical Study
Communications of the ACM XXVI(11):853-860, 1983

Bertram C. Bruce
Natural Communication between Person and Computer
in Strategies for Natural Language Processing
edited by Wendy Lehnert and Martin Ringle
Lawrence Erlbaum Associates, 1982

A. Bundy, Editor
Artificial Intelligence: an Introductory Course
Edinburgh University Press, 1978

C-L. Chang and R. C-T Lee
Symbolic Logic and Mechanical Theorem Proving
Academic Press, 1973

Peter P. Chen, Editor
Entity-Relationship Approach to Information Modeling and Analysis: Proceedings of the Second International Conference on Entity-Relationship Approach,
Washington, D.C., October 12-14, 1981 ER Institute, 1981

A Status Report on the Activities of the CODASYL
End User Facilities Committee (EUFC)
CODASYL End Users Facility Committee, 1979

Codd, E.F.
A Relational Model of Data for Large Shared Data Banks
Communications of the ACM, XIII(6), 1970

Codd, E.F.
How About Recently? (English Dialog with Relational Data Bases Using Rendezvous Version I)
in Shneiderman, 1978a

Marco Colombetti, Giovanni Guida, Barbara Pernici
and Marco Somalvico
Reasoning in Natural Language for Designing a Data Base
in Artificial and Human Intelligence, edited by
Alick Elithorn and Ranan Banerji
North-Holland, 1984

Irving M. Copi
Symbolic Logic, 4th Edition
Macmillan, 1973

Tom Crowe and John Jones
Potential Fallacies in the Design and Use of Data Bases
Computer Bulletin December 1974

C.J. Date and E.F. Codd
The Relational and Network Approaches:
Comparison of the Application Interfaces
Proceedings of the 1974 ACM-SIGFIDET Workshop,
ACM, New York, 1974

C.J. Date
An Introduction to Database Systems,
3rd Edition, Addison-Wesley, 1981

Alessandro D'Atri, Marina Moscarini and Nicolas Spyratos
Answering Queries in Relational Databases
SIGMOD Record XIII(4):173-177, 1983

Anthony Davey
Discourse Production
Edinburgh University Press, 1978

S. M. Deen
Fundamentals of Data Base Systems
MacMillan, 1977

and Peter Hammersley, editors
Databases: Proceedings of the 1st British National Conference on Databases
Pentech Press, London, 1981

Ivor Durham, David Lamb and James Saxe
Spelling Correction in User Interfaces
Communications of the ACM, XXVI(10), October 1983

Kemal Efe, Chris Miller and K. Hopper
The Kiwinet-Nicola Approach: Response Generation in a User-Friendly Interface.
Computer XVI(9):66-81, 1983.

R. A. Frost
SCHEMA: Yet Another Conceptual Schema Definition Language
Computer Journal XXVI(3):228-234, 1983

Sakti P. Ghosh
Data Base Organisation for Data Management
Academic Press, 1977

T.R.G. Green and H.T. Smith
Human Interaction with Computers
Academic Press, 1980

T.R.G. Green
Programming as a Cognitive Activity
in Human Interaction with Computers Green, 1980

David Greenblatt and Jerry Waxman
A Study of Three Database Query Languages
in Shneiderman, 1978a

Judith Greene
Thinking and Language
Methuen, 1975

P.A.V. Hall
Optimisation of a Single Relational Expression in a Relational Data Base System
IBM UK Scientific Centre Report UKSC0076 June 1975

Michael M. Hammer and Dennis J. Mcleod
Semantic Integrity in a Relational Data Base System
Proceedings of the International Conference on Very Large Data Bases:25-47, 1975

Michael Hammer
Error Detection in Data Base Systems
AFIPS Conference Proceedings XLV:795-801, 1976

L. R. Harris
User Oriented Data Base Query with
the ROBOT Natural Language Query System
International Journal of Man-Machine Studies IX:697-713, 1977

Wilfred Hodges
Logic
Penguin, 1977

C.A.R. Hoare
Programming: Sorcery or Science?
IEEE Software, I(2):5-16, April 1984

R.W.A. Hudson and J.A. Self
A Dialogue System to Teach Database Concepts
Computer Journal, XXV(1), 1982

A.T.F. Hutt
A Relational Data Base Management System
Wiley, 1979

S. Isard and H. C. Longuet-Higgins
Question-answering in English
Machine Intelligence VI:243-254, 19??

Philip C. Jackson
Introduction to Artificial Intelligence
Petrocelli Books, 1974

Jurgen M Janas
Towards More Informative User Interfaces
Proceedings of the Fifth International Conference
on Very Large Large Data Bases
IEEE, 1979

Matthias Jarke and Jurgen Koch
Range Nesting: A Fast Method to Evaluate Quantified Queries
SIGMOD Record XIII(4):196-206, 1983

Johnson-Laird, P.N.
Mental Models: towards a Cognitive Science of
Language, Inference, and Consciousness
Cambridge University Press, Cambridge, 1983

John D. Joyce and David R Warn
Command Use in a Relational Database System
AFIPS Conference Proceedings 52, 1983

S. J. Kaplan
Cooperative Responses from a Portable
Natural Language Query System
Artificial Antelligence XIX:165-187, 1982

William Kent
<u>Data and Reality</u>
North-Holland, New York, 1978

Margaret Kuhn and Ben Shneiderman
Two Experimental Comparisons of Relational
and Hierarchic Database Models
IFSM Technical Report No.31, Department of Information Systems Management, University of Maryland, 1978

Michel Lacroix and Alain Pirotte
Example Queries in Relational Languages
Technical Note N107
Manufacture Belge de Lampes et de Material Electronique
Research Laboratories, 1976

Michel Lacroix and Alain Pirotte
Domain-oriented Relational Languages
<u>Proceedings of the Third International Conference on Very Large Data Bases</u>, 1977

Frederick H. Lochovsky
Data Base Management System User Performance
Technical Report CSRG-90, Computer Systems Research Group, University of Toronto,T1977

Jim Longstaff, F. Poole and J. Roper
An alternative use of Natural Language for querying a relational database
<u>Proceeedings of ICMOD78</u>, Milan, 1978

Jim Longstaff
Query Specification and Accessing Strategies
for Relational Databases
PhD Thesis, Teesside Polytechnic, March 1980

Jim Longstaff, F. Poole and J. Roper
Teaching Relational Database Interactions
Using Natural Language Responses
in Deen and Hammersley, 1981

Frederick H. Lochovsky and Dionysios C. Tsichritzis
<u>Data Base Management Systems</u>
Academic Press, 1977

Frederick H. Lochovsky and Dionysios C. Tsichritzis
<u>Data Models</u>
Prentice-Hall, 1982

B.G.T. Lowden and R. Turner
An Introduction to the Formal Specification
of Relational Query Languages
<u>The Computer Journal</u>, XXVIII(:2), 1985

James D. McCawley.
Everything that Linguists have Always Wanted to Know about Logic
University of Chicago Press, 1981

Kathleen R McKeown
Discourse Strategies for Generating Natural Language Text
Artificial Intelligence 27 (1985) 1-41

James Martin
Computer Data-Base Organization, 2nd Edition
Prentice-Hall, 1977

M. Missikoff and M. Scholl
Relational Queries in a Domain Based DBMS
SIGMOD Record XIII(4):219-227, 1983

Glenford J Myers
The Art of Software Testing
John Wiley and Sons, 1979

E.S. Page and L.B. Wilson
An Introduction to Computational Combinatorics
Cambridge University Press, 1979

J. L. Peterson
Computer programs for detecting and correcting spelling errors
Communications of the ACM XXIII(12), 1980

C. J. Prenner
A Uniform Notation for Expressing Queries
Memorandum No. UCB/ERL M77/60
Electronics Research Laboratory
University of California, Berkeley, 1977

Phyllis Reisner
Use of Psychological Experimentation as an Aid to Query Language Design
IEEE Transactions on Software Engineering
SE-3(3):218-229, May 1977

Allison Rebeca Reuber
An Assessment of the Ability of the Extended Relational Model to Serve as a Conceptual Schema Language
Master's Thesis, Queen's University, Ontario, 1981

Elaine Rich
Natural Language Interfaces
Computer, XVII:9, September 1984

Nicholas D. Roussopoulos
A Semantic Network Model of Data Bases
PhD Thesis, University of Toronto, 1976

Sandra B. Salazar and Gerald A. Wilson
A System for Interactive Error Detection
Proceedings of the Fifth International Conference on Very Large Data Bases
IEEE, 1979

Geoffrey Sampson
The Form of Language
Weidenfeld and Nicolson, 1975

Barbara C. Sangster
Natural Language Dialogue with Data Base Systems: Designing for the Medical Environment
Information Technology, J.Moneta (editor):183-187, 1978

H.A. Schmid and J.R. Swenson
On the Semantics of the Relational Data Model
Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data
ACM, 1975

G. C. H. Sharman
Update-By-Dialogue: an Interactive Approach to Database Modification
SIGMOD Journal Proceedings of the International Conference on the Management of Data:21-29, 3-5 August 1977

Ben Shneiderman, Editor
Databases: Improving Usability and Responsiveness
Academic Press, 1978a

Ben Shneiderman
Improving the Human Factors Aspect of Database Interactions
ACM Transactions on Database Systems, 3(4), December, 1978b

John F. Sowa
Conceptual Structures
Addison-Wesley, 1984

Roger Tagg
Query Languages for some Current DBMS
in Deen and Hammersley, 1981

S.J.P.Todd
The Peterlee Relational Test Vehicle -- a System Overview
IBM Systems Journal XV(4), 1976

E. Vandijck
Towards a More Familiar Relational Retrieval Language
Information Systems 2:161-165 (1977)

Adrian Walker
Automatic Generation of Explanations
of Results from Knowledge Bases
IBM Research Report RJ3481, 1982

Gio Wiederhold
Database Design
McGraw-Hill, 1977

Zloof, M.M.
Design Aspects of the Query-by-Example
Data Base Management Language
in Shneiderman, 1978a

DATABASES

Proceedings of the 1st British National Conference on Databases held at Jesus College Cambridge, 13-14 July 1981

Edited by

S M Deen, *University of Aberdeen*
P Hammersley, *Middlesex Polytechnic*

PENTECH PRESS

London: Plymouth

Contents

QUERY VALIDATION: REVERSE TRANSLATION AND THE CONNECTION AND SELECTION TRAP

T Crowe, D R Hainline, R G Johnson

Thames Polytechnic

This paper describes some aspects of research into the validation of queries using Reverse Translation and in particular how the problems of the connection trap and selection trap can be overcome.

QUEVAL is a prototype software system developed to validate queries and so help naive end-users to assert the validity of their queries when using a query language. The development of QUEVAL necessitated the development of a relational database and its associated relational query language on a DEC-10 system. This paper explores in particular the use of Reverse Translation for the connection and selection operations.

1. INTRODUCTION

Query languages are an increasingly important end-user facility in the use of DBMS for the retrieval of information. This paper describes some aspects of research into the validation of such queries using Reverse Translation (RT) and in particular how RT can be used to overcome the problem of the connection trap as described by Codd and the selection trap.

Traditionally the concept of validation has been applied to the situation where raw data is validated prior to its use by tested and proven programs. With the increasing use of databases the reverse is likely to be the case. One-off queries, ie unproven programs will access the database of validated data. End-users will need help in asserting the validity of such programs.

The approach taken in this research programme was to build a prototype software system, QUEVAL, to effect such validation. To this end a relational database with its associated relational algebraic query language, has been developed on a DEC-10 system. To date all the relational algebraic operations, with the exception of division, have been implemented. The relational approach was used because it offered a formalism appropriate to research and the query language is well documented in the literature. There are many techniques that can be applied to help in the validation of a query and it is planned in this research programme to investigate them by implementing them in stages. The first technique to be implemented was that of RT.

A user who has initiated the query is able to see that meaning explicated in a set of natural language sentences generated by the validation system. This process of understanding extends to the final relation which is the answer to the query. In fact, the query in declarative form will be one of the many possible valid descriptions of the final relation. This provides a method of checking the semantic validity of a query expression: the user matches his understanding of the derived relations against the machine.

There are many interesting aspects to this research work, particularly in the 'data dictionary' area where it is necessary to have graph structures to support the RT process. However, this paper deals with the connection trap and the selection trap and how they can be overcome using RT.

Codd [1] describes the connection trap, where a potentially misleading meaning can be attributed to a relation resulting from the re-joining of the projected components of an all-key relation. (RT can call attention to the difference in meaning). An equally misleading result can arise from syntactically valid selection operations on certain relations.

2. VALIDATION

The error prone and expensive part of any data processing system is usually the inclusion of new components, whether these be new programs that must be written and tested or data that has to be captured and validated. In the context of traditional data processing the process of validation of data is reasonably well understood [2]. [see Figure 1].

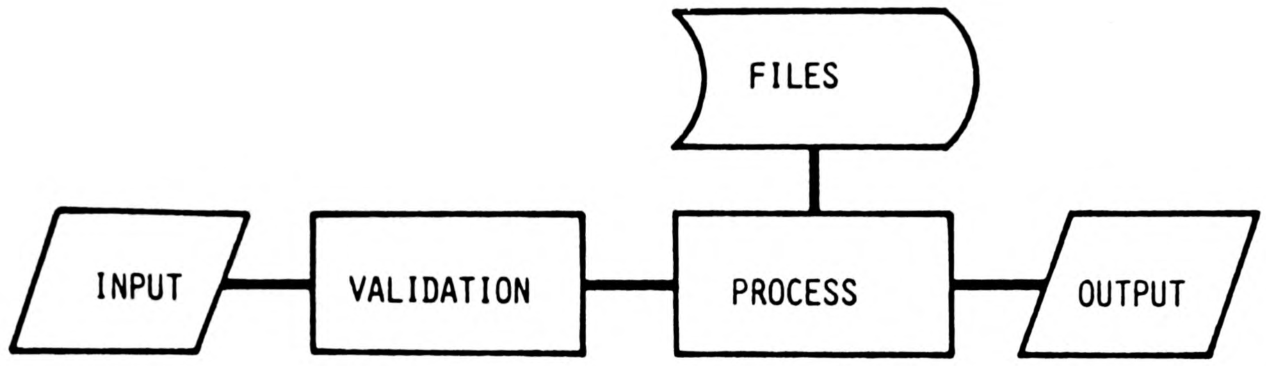


Fig. 1

Our research has been concerned with the validation, not of the data but of programs and, in particular, queries.

Figure 2 illustrates a typical systems environment in which a query language would be used. In such a situation, the queries are novel and unproven whilst the data tends to be static and correct (for the intended query).

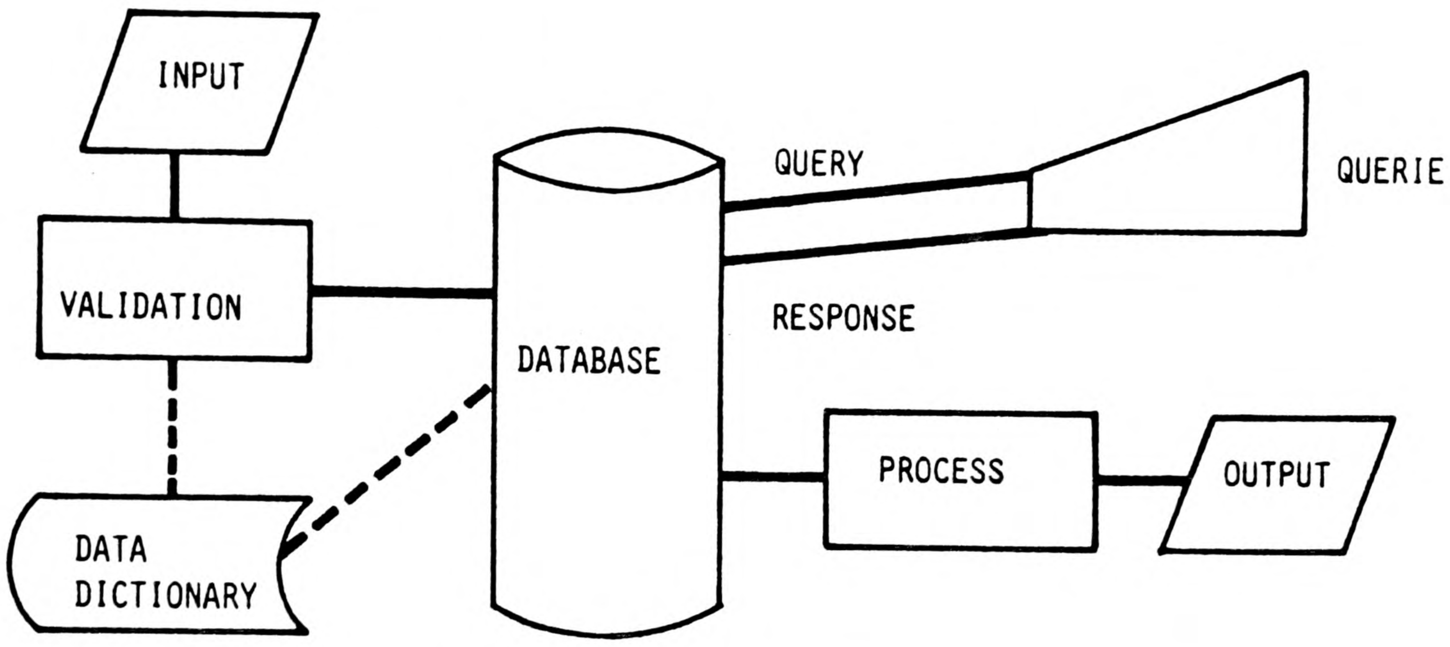


Fig. 2

It is thus the query (ie the program) that needs to be verified. This is the converse of the batch data validation and is rapidly becoming more common as data is collected into integrated databases and made accessible to new groups of users through Management Information Systems.

In this situation, the database user is a programmer; his query will be executed, and the results produced. However, the classical methods of validating programs are not available. The query is unlikely to be tested with sets of test data. There is no previous system, acting as a standard, with which to compare the output. There is no user community quick to spot anomalies in the output. The queries created by the competent casual user will require a different approach to validation, if they are to be validated at all.

3. THE RESEARCH APPROACH

The approach taken in this research has been both developmental and theoretical. QUEVAL, a software package, has been developed to provide an end user facility for the validation of queries in the belief that the development of such a package would generate theoretical problems and lead to a better understanding of errors inherent in the query process. QUEVAL has a relational database which stores and manipulates the data.

The Relational approach was adopted in this research because it is a formal approach that is both well understood and well documented.

Following the authors' experience with PRTV [3], a similar but simplified relational algebra query language was used. The authors feel that the relational algebra system could also be considered as a model of a range of other query languages.

The symbols used in our relational algebra expressions and used in examples in this paper are:

-	Difference	*	Natural join
%	Projection	+	Union
:	Selection	.	Intersection
<-	Assignment	->	Rename of entity

There are many areas of error control that could be researched, ranging from the well understood techniques of syntax error detection to an understanding of the propensity of humans to make errors. This paper concentrates on one aspect of this range that is the use of Reverse Translation (RT). The computer system states in English what it understands the query to be by giving a description in English of the generated answer. In fact, the query in declarative form will be one of the many possible valid descriptions of the final relation. This provides a method for checking the semantic validity of a query expression: the user matches his understanding of the derived relation against the machine's Reverse Translation.

The pioneering effort in this field is, of course, the RENDEZVOUS system being developed by E F Codd and others, [4] which includes natural language generation as part of dialogue with the user to enable him to formulate his query precisely in natural language.

E F Codd notes that "... very little work on the generation of natural language has been published... and none of this appears to treat the problem in the context of query formulation". Recent work in this country includes a system which gives a narrative description of any noughts-and-crosses game [5], and, closest to our own approach, the system being developed by Longstaff, Poole and Roper [6] which aims at reverse translation into natural language of queries formulated in a relational calculus - like language.

The provision of reverse translation of query language expressions requires two things: the existence of a data structure capable of capturing the aspects of meaning not derivable directly from a relation, and the existence of an "analyser" capable of generating the most comprehensible natural language expression from this data structure. These two requirements are in large part separable. An adequate solution to the first problem in no way guarantees an adequate solution to the second.

4. THE MEANING OF A RELATION

The relations in QUEVAL are in first normal form and the meaning of a relation is not deducable from a knowledge of the domains.

A database does not only contain data but usually has data about the data, ie a meta database, the data dictionary. In QUEVAL we have used the data dictionary to contain a clear understanding of meaning of the relations and by modest additions to assist in the RT process

In QUEVAL, in addition to the usual information in the data dictionary for each domain, there is stored the singular and plural forms of the name of each entity of which the domain values are representatives. In addition it is recorded whether the entity is animate or inanimate.

For example:

Domain name:	EMP
Singular entity name:	EMPLOYEE
Plural entity name:	EMPLOYEES
Animate:	YES

For each attribute in a relation its domain is recorded. All the relationships embedded in each relation are recorded. For each relationship, the domain which fills each argument of the relationship
is recorded and the attribute in the relation that represents that role.

For example:

```
EMPLOYEE (EMP, EMPNAM,   LANG,  EMPGDE)
```

There is a many to one relationship of speaking as a native language between the employee and the native language:

Relationship:	SPEAKS AS A NATIVE
Argument 1:	EMP
Argument 2:	LANG
Complexity:	MANY TO 1

Thus the relation A, shown in Fig. 3A, gives rise to the following piece of text, which is expressed as a description of a single tuple in the relation.

Attribute name	EMP	LANG
	A003	French
	A007	English
	A010	English
	A016	French
	A027	German
	A030	English

Fig. 3A Relation A

Thus QUEVAL can generate an English description of relation A:

> The indicated EMPLOYEE (EMP) and possibly others, speaks as a native the indicated LANGUAGE (LANG) and it only.

This sentence it can be seen has been generated from the data in the data dictionary. Suppose now that a slightly different design had been adopted in which all languages spoken by employees were recorded (Relation B, Fig. 3B). The complexity would have been MANY TO MANY and the resulting text would have been:

> The indicated EMPLOYEE (EMP) and possibly others, speaks the indicated LANGUAGE (LANG) and possibly others.

Complexity: N:M

Relation B

Attribute	EMP	LANG
	A611	English
	B021	English
	B021	French
	C512	German
	C512	French
	C512	English

Fig. 3B - Relation B

5. AN EXAMPLE OF A RELATIONAL OPERATION

The full range of operations is described in Johnson [7]. When the user makes a query he manipulates one or more relations. QUEVAL manipulates the relations and the resulting tuple descriptions in parallel. Thus taking the relation B in Fig. 3B consider the simple operation of projection.

```
B1 ← B [LANG]
```

The resulting relation is shown in Fig. 4 and lists the languages spoken by employees.

Attribute name:	LANG
	French
	English
	German

Fig. 4 Relation B1

The text that would be produced from this query is:

At least one EMPLOYEE speaks the indicated LANGUAGE (LANG), and possibly others.

As can be seen with the above example, reference to the attributes which have been projected out must not simply disappear from the reverse translation for that would not be faithful to the actual meaning of the derived relation. For the relation B1 in Fig. 4, it is not known which employees have learned these languages, but the history of this relation proves that for each of these languages, at least one employee can be found who has learned it.

6. THE SELECTION TRAP

E F Codd [1] described the "connection trap", where a potentially misleading meaning can be attributed to a relation resulting from the re-joining of the projected components of an all-key relation. (We will show later how RT can help to protect against such a trap).

An equally misleading result can arise from syntactically valid selection operations on certain relations.

Consider a user wishing to find employees whose native language is French. He would receive the desired information from the following query:

```
F1 ← (A: [LANG = "French"]) % [EMP]
```

To find employees whose native language is not French, a symmetrical expression is used:

```
     F2  ←  (A: [LANG ≠ "French"]) % [EMP]
```

This slightly more complicated query, which includes a negation, has an algebraic expression which parallels the previous, positive, query, with "≠" substituted for "=". This is probably the relationship that most casual users would expect to exist between the two queries, and their algebraic expressions, and they would in this case be correct.

Now suppose the user wants the employee numbers of all employees who have learned French. He again uses a selection followed by a projection:

```
     F3  ←  (B: [LANG = "French"]) % [EMP]
```

If the user wants the employee numbers of all employees who have not learned French, a request apparently symmetrical to the previous request, he may be tempted to use the following:

```
     F4  ←  (B: [LANG≠"French"]) % [EMP]
```

This formulation is syntactically valid, and does not involve a violation of data constraints. The resulting relation will contain many employee numbers, and has a coherent meaning. However, it was not what was intended.

The user has obtained, not employees who have not learned French, but employees who have learned one or more languages other than French.

A user using the reverse translation facility to check the previous two queries would have received the following interpretation of F2

> The indicated EMPLOYEE, (EMP) and possibly
> others, speaks as a native a LANGUAGE which
> is not French, and it alone.

F4 would have been rendered

> The indicated EMPLOYEE, (EMP) and possibly
> others, speaks a LANGUAGE which is not
> French, and possibly others.

The latter translation is at least a flag on the selection trap. More can be done: the natural language rendering of the complement of the selection expression can be incorporated into the "and possibly others" phrase which follows a selection translation in a complex relationship. This would yield a translation of the last phrase above as "... the indicated LANGUAGE, which is not French, and possibly others, including French".

To correctly find employees who have not learned French, the user first creates a relation consisting of those who have done so, and then differences it from a relation containing all employees who have learned languages.

```
    F5 ← (B% [EMP] ) - ( (B: [LANG = 'French'] ) % [EMP])
```

THE RT of F5:

> The indicated EMPLOYEE (EMP) and possibly
> others, speaks one or more LANGUAGES,
> but does not speak a LANGUAGE which is French.

7. THE CONNECTION TRAP

It is useful to see how RT might illuminate "connection trap" problems.

These can arise when components of a relation previously projected out are re-joined, resulting in a relation whose "surface structure" is identical to the original relation, but whose underlying meaning is different.

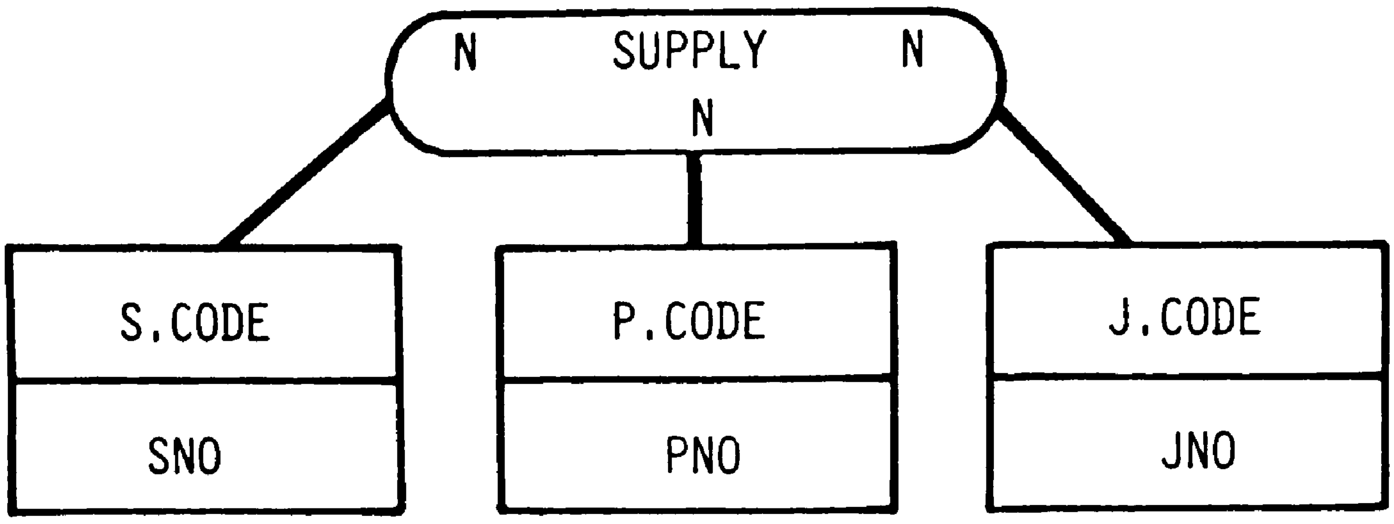
The following relation and series of operations illustrates this well-known problem.

Complexity:	N:N:N		
Relation	S		
Domain	S.CODE (Supplier)	P.CODE (Part)	J.CODE (Job)
Attribute	SNO	PNO	JNO
	S1	P1	J1
	S1	P1	J2
	S2	P1	J1

The intended meaning of this relation is that supplier SNO supplies part PNO to job JNO.

A graphical representation of the relation.

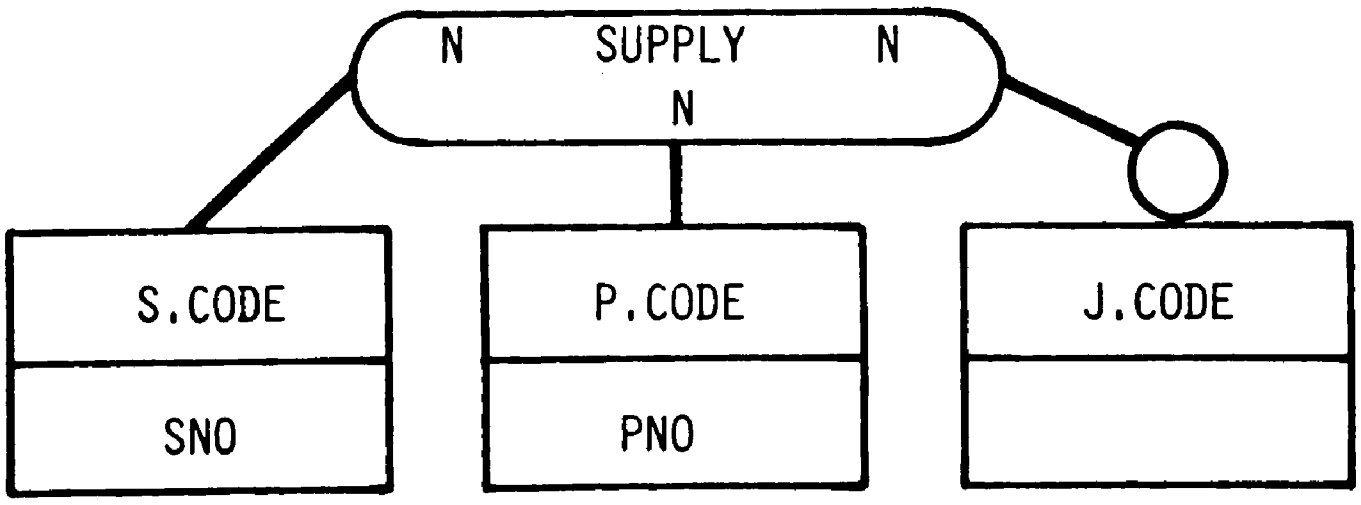
S:



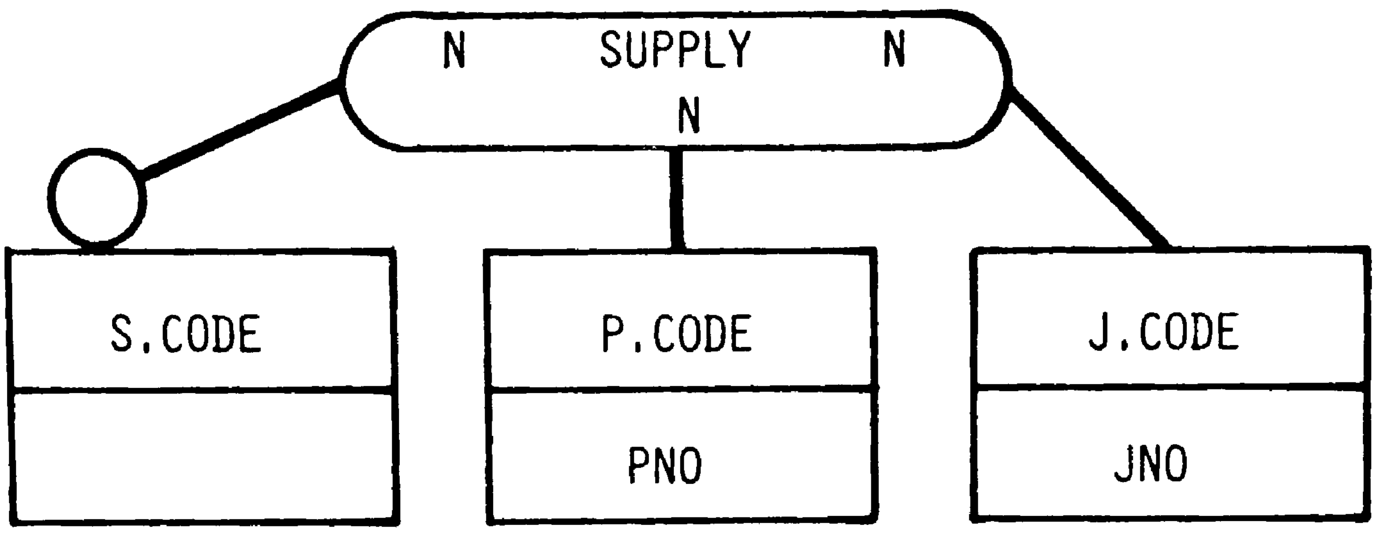
Let

```
S1 ← Supply % [SNO,PNO]
S2 ← Supply % [PNO,JNO]


Complexity:    N:N

Relation       S1
               S.CODE        P.CODE
Domain

Attribute      SNO           PNO

               S1            P1
               S2            P1
```

S1:



Complexity:	N:N	
Relation	S2	
Domain	S.CODE	P.CODE
Attribute	PNO	JNO
	P1	J1
	P1	J2

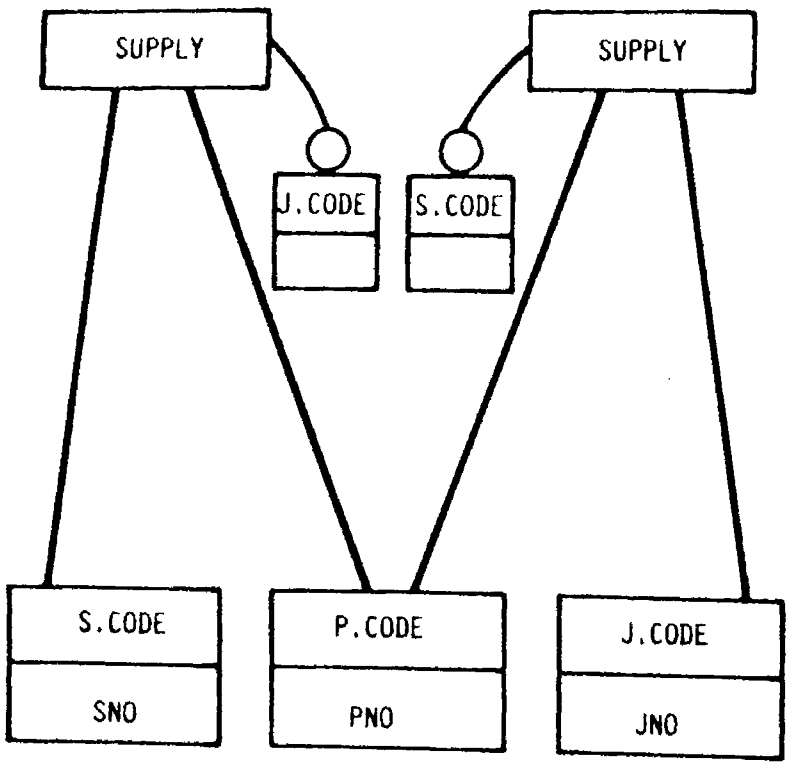
S2: 

The Connection Trap can arise when S1 and S2 are re-joined:

$S3 \leftarrow S1 * S2$

Complexity:	N:N:N		
Relation	S3		
Domain	S.CODE	P.CODE	J.CODE
Attribute	SNO	PNO	JNO
	S1	P1	J1
	S1	P1	J2
	S2	P1	J1
	S2	P1	J2

A graphical representation of S3:



The RT of S:

> The indicated SUPPLIER (SNO), and possibly others, supplies the indicated PARTS (PNO), and possibly others, to the indicated JOB (JNO) and possibly others.

The RT of S3:

> The indicated SUPPLIER (SNO), and possibly others, supplies the indicated PART (PNO), and possibly others, to at least one JOB.
>
> One or more SUPPLIERS supplies the indicated PART (PNO), and possibly others, to the indicated JOB (JNO), and possibly others.

The actual relationships embodied in the two relations, S1 and S3, are made explicit by the RT of each.

Of course, the availability of an RT, or of any other device for explicating the underlying structure of a relation cannot guarantee a correct user understanding of a relation's meaning.

8. CONCLUSIONS

The authors believe that use of reverse translation provides a significant advance in the development of query languages. While inexperienced users have a higher error rate, all users can benefit from the opportunity provided of comparing their intuition for a particular query with an alternative textual formulation of that query.

The research on this paper, together with the other research mentioned, confirms the authors in their belief that reverse translation is both necessary and valuable for query language systems. This paper has shown the text produced from relational algebra expressions. No previous work known to the authors has given detailed consideration to this problem.

The system described here is a prototype system and further research is needed to improve the natural language text and to fully develop other facilities. However, the authors believe the present system represents an advance on other query systems currently available.

Query languages are not designed specifically to avoid errors in the formulation of the query, except insofar as good design inherently must serve that end, and it could be that this research would lead to a reconsideration of the design of query languages or more possibly, query systems.

9. ACKNOWLEDGEMENT

The authors would like to express their thanks to Dr B Aldred of IBM Scientific Centre, Winchester for his helpful discussions and encouragement during this project.

10. REFERENCES

[1] E F Codd - A relational model of data for large shared data banks, Comm ACM 13, June, 1970

[2] T Crowe and J H Jones - Fundamental Nature of errors in data capture; and the use of data dictionary, Management Datamatics IAG, Amsterdam) 4,3, 1975

[3] S J P Todd - The Peterlee Relational Test Vehicle, IBM Syst J 15,4 1976

[4] E F Codd - How about recently? (English dialogue with relational databases using Rendezvous Version 1), in Schneiderman, B Databases: Improving Usability and Responsiveness, Academic Press, 1978

[5] A Davey - Discourse Production, Edinburgh University Press, 1978

[6] J Longstaff, F Poole and J Roper - An alternative use of Natural Language for querying a relational database, ICMOD78, Milan, 1978

[7] R G Johnson, T Crowe and D R Hainline - Verifying Queries from Naive End-users, to be published.

APPENDIX II

Reverse Translation Grammar

A Reverse Translation Graph has the following structure:

```
RT ::= Entity-Node-RT
      + {Predication-Node-RT + Entity-Node-RT} 1..n

Entity-Node-RT ::= Indicator-Phrase
                 + Domain-Reference
                 + ( Attribute-Name )
                 + ( Role-Phrase )
                 + ( Qualification-Phrase )
                 + ( EDegree-Phrase )

Indicator-Phrase        ::= [ Participating-Phrase | Non-Participating-Phrase ]
Domain-Reference        ::= name of domain this node corresponds to
Attribute-Name          ::= name of attribute this node corresponds to
Role-Phrase             ::= "acting as a" + role-name
Qualification-Phrase    ::= Introducer + Qualifier-tree
EDegree-Phrase          ::= [ N-phrase | 1-phrase ]

Predication-Node-RT     ::= [ Main-Predication-Phrase | Case-Indicator-Phrase ]
Main-Predication-Phrase ::= [ Positive-Predication-Phrase |
                              Negative-Predication-Phrase ]
N-phrase                ::= "and possibly others"
1-phrase                ::= "and it alone"
Introducer              ::= [ Representer-phrase | Rel-pro ]
Representer-phrase      ::= "whose" + Representative-Name
Rel-pro                 ::= [ "which" | "who" ]
Participating-Phrase    ::= "the indicated"
Non-Participating-Phrase::= [ Principal-Entity-Phrase | Non-Principal-Entity-Phrase ]
Principal-Entity-Phrase ::= "At least one"
Non-Principal-Entity-Phrase::= [ UnQual-Phrase | Qualif-Phrase ]
Unqual-Phrase           ::= "that"
Qualif-Phrase           ::= "those"
Qualifier-tree          ::= [Simple logical sub-graph | RT
```

APPENDIX III

Program Listing

```
PROGRAM revTrans(INPUT,OUTPUT, F);
CONST
 strmax  = 20; {max length of any string}
 maxreal = 536870911.0;{scientific notation for all higher values}
 menumax   = 20;{used in menu routines -- max choices permitted}
 stackmax  ,= 24; {max number of Enodes which can be uttered}
 maxvalues  = 16;
 degmax     = 6; {max degree of any relation}
 attmax     =  8; {max number of columns in any relation}
 maxpreds  .=  1; {max number of root  predications a relation can have}
 permmax    =  6; {Must always be  factorial of largest base relation}
                  {degree permitted}
 dmax       =  6; {max number of entity/domains allowed in data dictionary }
 rmax       = 16; {max number of relations -- base and derived -- allowed}

                  {The following are synonyms used in type definitions
                   They are the equivalent of user-defined enumerated
                   types. However, unlike enumerated types, they can
                   be input and output to files/the screen, using
                   standard read and write procedures, which saves writing a
                   special I/O procedure for each type (at the expense
                   of a certain amount of built-in error checking)    }

 no             = 0 ;{ my own read/writeable boolean type }
 yes            = 1 ;

 negative       = 0 ; { for 'sense' (q.v.) of a predication }
 positive       = 1 ;

{ ℓattribute    = 0;} { for future research -- seeing if 'entity-entity' }
{ relationship  = 1;} { relations should be translated differently from }
                      {entity-attribute predications}

 singular       = 0 ;{ to mark those elements of RT which need the  }
 plural         = 1 ; { distinction }

 exported       = 0;{ 'Edegree' (q.v.) values }
 one            = 1; { Only the middle two used in present system }
 many           = 2;
 all            = 3;

 ent_att        = 0;
 ent_ent        = 1;

 nom            = 0; { markers for case }
 obj            = 1;

 inanimate      = 0; { markers for who/which distinction }
 animate        = 1;

 Bool           = 0; { the different data types allowed in current system }
 int            = 1;
 rel            = 2;
 Cha            = 3;
 Str            = 4;
 Cat            = 5;  {for future expansion}

 ropnode        = 1; { the different node types }
 copnode        = 2;
 pnode          = 3;
 enode          = 4;
 valuenode      = 5;

 orx            = 0;  { the values a Logical Node (ropnode) can take }
 andx           = 1;
 notx           = 2;

 Stℓ            = 1; { the different values a Comparison Node can take }
 Seℓ            = 2;
 Negℓ           = 3;
 Eqℓ            = 4;
 Geℓ            = 5;
 Gtℓ            = 6;
 Ssℓ            = 7;

{ℓP}
TYPE
                           { when two Simple Logical Sub-Graphs are presented
                             as candidates for composition, these are the possible
```

```
                    values which the proceaure Qreduce can generate }
   result_type = (empty_set, universal_set, left, right, either,
                  compose, newop, incompatible_operators);

   File_name_type = ARRAY [ 1..32] OF CHAR; { PRIMOS definition of
                                                a path-name }

   StColType = RECORD { holds attribute names for relational operations }
                 count: 0..attmax;
                 names: ARRAY[1..attmax] OF string[strmax];
               END;

   animate_type = inanimate..animate; { see explanations for each set}
   number_Type  = singular..plural;   { of values, above }
   case_type    = nom..obj;

   Domindex_type  = 1..dmax;    { references to domain definitions }
   relindex_type  = 1..rmax;    { ditto for relations }
   col_index_type = 1..degmax;  { ditto for attributes }
   cols_type      = RECORD { holds references to attributes }
                      Num: col_index_type;
                      cols: ARRAY[col_index_type] OF col_index_type;
                    END;
{  Visit_count_type = 0 .. 12;}{ not used in current system }

  att_index_type    = 1..attmax;
  perm_type         = 1..Permmax;
  Edegree_type      = exported .. all;

  menu_Choice_type = 0..menumax;
  menu_type = RECORD
                Choices : menu_Choice_type;
                Options : ARRAY [ menu_Choice_type] OF string[strmax];
              END;

  p_table_type  =  ARRAY [ 1..24, 1..4 ] OF  integer;{ permutation table}
  pro_table_type = ARRAY [ inanimate..animate, nom..obj] OF string[strmax];

  yesno_type  = no .. yes ;

{Here are all the type definitions relating to the RT apparatus}


  nodepointer = ^node;
  node_id_type  = 0..maxint;{each node has unique id}

  pointerstack = RECORD
                    nf : 1..stackmax;
                    pointers : ARRAY[ 1.. stackmax] OF nodepointer;
                 END;
  nodetype =  ropnode .. valuenode;
  setofnodetypes = SET OF nodetype;
  setofchar      = SET OF char;

  roptype = orx .. notx ; { Logical Node type }
  coptype = St£ .. Ss£  ; { Comparison Node type }


  Sense_type  = negative .. positive;


  phrase_type = RECORD { for holding predication phrase strings }
                  Main: yesno_type;
                  CASE Main:yesno_type  OF
                   no:  ( case_phrase : String[strmax]);
                   yes: ( Alt_phrases : ARRAY[Sense_type, number_type]
                                        OF String[strmax]);
                END;

  Sentence_type   = ARRAY [ 1..degmax] OF phrase_type;

  node = RECORD
         descendants : 1..degmax; {number of pointers not NIL}
         ptr         : ARRAY [1..degmax] OF nodepointer;
         typeofnode  : nodetype;

         CASE typeofnode:nodetype OF
           ropnode : (Op:roptype);
           copnode : (cop:coptype;
                      cdegree: Edegree_type; )
```

```
          pnode    : (Pred_id : 1..rmax; (unique predication identifier)
                         pperm    : Perm_type : (which permutation it is)
                                       { next  field not used in
                                         current implementation }
                         pred_type : ent_att..ent_ent; (some predications of
                                                 degree two are entity/
                                                 attribute ones)
                         psense   : Sense_type;
                         pnumber  : singular .. plural;
                         Alt_sentences  : ARRAY [perm_type] OF Sentence_type);
         enode    : (
                     attnum      : 0 .. degmax; ( attribute  entity refers to)
                     entdomain_id : domindex_type; (domain of this argument)
                     dummy       : yesno_type; { for future research --
                                     for implementing predication lowering}
                     edegree     : edegree_type;
                     node_id     : node_id_type;
                     role        : ARRAY [ Number_type] OF  string[strmax];
                     trimmed     : yesno_type;{ for future research --
                                     for implementing user re-definition of
                                     verbose reverse translations }
                     shortdesc : string[strmax]); { ditto above }
          valuenode: (value:string[strmax]);
         END; {of *node*}


(These are the data Dictionary types which hold information about
 each relation -- both base relations and derived relations -- and
 each domain   -- note that Domain is synonymous with entity as far
 as reverse translation is concerned.)

attHdrtype = RECORD  (info. on each attribute in a relation)
              aname  : string[strmax];
              Attdomain_id : 1..dmax; (location  of  domain in  data dict)
             END;

rhdrtype = RECORD
             Rname      : string[strmax];
             Rdegree    : 1..degmax;
             Key_cols   : cols_type;
             Rperm      : Perm_type;
             attribute  : ARRAY[1..attmax] OF attHdrtype;
             preds      : 1..maxPreds; (every relation has at least one)
             ptr        : ARRAY [1..maxPreds] OF nodepointer;
            END;

DomDictype = { info. on each domain }
  RECORD
    entity_set_name    : ARRAY [ number_type ] OF string[strmax];

    animate_value  : inanimate .. animate;
    Self_ID        : yesno_type;

    repdata : RECORD
       represented : yesno_type;
       CASE represented : yesno_type  OF
         no: ( );
         yes:( repName : ARRAY [ singular  .. plural] OF string[strmax]);
       END; (of repdata record)

    Rangedata  : RECORD
      datatype : Bool .. dat;
      CASE datatype: Bool .. dat  OF
            Bool, Cha, Str  : ( );
            rel             : (rmin, rmax : REAL);
            int             :(imin, imax : integer );
       END;(of Rangedata record)

    orderdata  : RECORD
      ordered : yesno_type;
      compphrases : ARRAY [St£..Ss£] OF string[strmax];
     END; (of orderdata)

    valuesetdata : RECORD
       Limited : yesno_type;
         CASE Limited: yesno_type  OF
           no: ( );
          yes: ( values : RECORD
                          count :     1..maxvalues;
```

```
                                  Stringf : ARRAY  [ 1..maxvalues]
                                                OF string [strmax] ;
                                          END; {of values record}};
          END; {of valuesetdata}

     END;{ of RECORD}

     rd_type = ARRAY [1..rmax] OF rhortype;
     dd_type = ARRAY [1..dmax] OF DomDictype;

     {end of TYRE definitions}

     {{P}
     VAR
       dd : dd_type;
       gsense   : sense_type; {for controlling determinant phrases}
       gsuppress: boolean; {for suppressing utterance of entity names
                           and relative pronouns when the entity is
                           self-identifying and there are values present}
       glevel : 0 .. 24; {controls indentation depth of RT phrases}
       gprojected_out: boolean; {controls comparison phrases}
       attribute_names : StColType;
       operation       : char;
       cols            :  cols_type;
       midnodestack, leftnodestack, rightnodestack : pointerstack;
       p_table        : P_table_type;   { hold permutations }
       pro_table      : Pro_table_type; {holds pronouns }
       row, col       : integer;{ row and column index variables for
                                  loading tables }
       I              : 0 .. maxint; {loop control variable}
       debug          : boolean; { to switch debugging statements on and off }
       Q              : menu_Choice_type; {holds user's menu choice}
       rd             : rd_type; { holds relation headers }
       rdnf, ddnf     : integer; {next free location in rd and dd}
       newr           : integer;{holds relation location in menu choice 4}
       next_node_id   :node_id_type;

       f              : TEXT ; { our all-purpose file variable}
       f_name         : File_name_type;
       Main_menu,
       Ops_menu,
       datatype_menu : menu_type;

       r               : relindex_type;
       d               : domindex_type; {dummy variable for initial call of
                                              utter}

     {{P}
     %include'HD12XT>personal>extrprocs';
     {%include'HD12XT>personal>p_qreduce';}
     %include'HD12XT>personal>p_utter';
     %include'HD12XT>personal>p_select';
     %include'HD12XT>personal>p_setops';
     %include'HD12XT>personal>p_permute';
     %include'HD12XT>personal>p_join';
     %include'HD12XT>personal>p_Project';
     %include'HD12XT>personal>p_command';


     {{P}
     BEGIN  {main body}
        debug := BQuery ('shall I switch the debugging facility on?');

        reset(F,'TABLE.PRONOUNS');
        FOR row:= inanimate TO animate DO
         FOR col:= nom TO all DO     BEGIN
           readln(F, pro_table[row,col]);
           IF debug THEN
              writeln('row, col, pronoun =', row, col, pro_table[row, col]);
                                                END;
        Close(F);

        reset (F, 'TABLE.PERMS');
        FOR row := 1 TO 24 DO
          FOR col := 1 TO 4 DO
            READ (F, p_table [row,col]);
        close(F);

        reset (F, 'MENU.MAIN');
```

```
   load_menu (F,Main_menu);
   close(F);

   reset(F, 'MENU.OPS');
   load_menu(F, Ops_menu);
   close(F);

   reset '( F, 'MENU.DATATYPES');
   load_menu (F, datatype_menu);
   close (F);

   get_file_name(F_name  );
   reset (F, F_name);
   load_dict(f);
   close (F);

{{P}
   REPEAT
     show_menu(Main_menu);
     Q := Iquery('Your choice ', 0, main_menu.choices-1);

     CASE Q OF
       0: BEGIN  END;

       1: create_domain;

       2: create_relation;

       3:BEGIN
          REPEAT
             newr:=rdnf;
             writeln;
             glevel:=0; gsense:=positive;  gsuppress:=FALSE;
             gprojected_out:=FALSE;
             processcommand;
             writeln;
             IF newr < rdnf THEN BEGIN  show_relation( newr);
                                        writeln;
                                        utter( rd[newr].ptr[1], newr, 0)  END;
             writeln;
          UNTIL NOT (BQuery('Another command?'));
          END;

      4: IF ddnf>1 THEN
             FOR i:= 1 TO ddnf-1 DO BEGIN   show_domain( i); pause END
           ELSE writeln ('no domains defined as yet.');

      5: IF rdnf>1 THEN
            FOR R:= 1 TO rdnf-1 DO                BEGIN
              show_relation (r);
              writeln;
              writeln('=======================================');
              writeln;
              writeln('Reverse Translation: ');
              writeln;
              glevel:=0; gsense:=positive;  gsuppress:=FALSE;
              gprojected_out:=FALSE;
              midnodestack.nf:=1; {reset}
              utter ( rd[R].ptr[1], R , 0);
              writeln; pause                            END
          ELSE writeln('no relations defined as yet.');

        6: BEGIN  writeln('Switching debug status.');
                  debug := NOT debug;
             END;

        otherwise  writeln('choose 0 to ', main_menu.choices-1:3);
     END {of CASE}
   UNTIL (Q = 0);

   rewrite(F, 'DATA.lastrun');
   dump_dict(F);
   close (F);

   END.
PROCEDURE attach( op: optype; VAR p1: nodepointer;
                                  p2 : nodepointer);
{attaches a copy of tree pointedto by p2 to p1 }
```

```
            IF ptr[1]^.descendants = 0 THEN  {no graph already attached}
            BEGIN
              ptr[1]^.descendants := 1 ; {there will be now}
              NEW(ptr[1]^.ptr[1], copnode);
               WITH ptr[1]^.ptr[1]^ DO BEGIN  {with the node we've just made}
                  cop:=XOp;
                  {Edegree :=                 }
                  descendants := 1 ;
                  IF debug THEN BEGIN
                    writeln(' we have created a comp.op node, whose value is ', cop);
                  END;
                  typeofnode := copnode;
                  NEW (ptr[1], valuenode);{now create a Valuenode, pointed to by the copnode}
                  s1 := Svalue; {necessary step due to PRIME bug}
                  ptr[1]^.value := s1;
                  ptr[1]^.descendants := 0;
                  ptr[1]^.typeofnode := valuenode              END
             END
             ELSE
             BEGIN   {there is already a QGraph}
               oldgraph := pointers[nf-1]^.ptr[1]^.ptr[1]; {store temporary}
               NEW(ptr[1]^.ptr[1], Ropnode);
               WITH ptr[1]^.ptr[1]^ DO BEGIN
                  op := andx;
                  typeofnode:=ropnode;
                  descendants := 2;
                  ptr[1] := oldgraph;
                  NEW(ptr[2], copnode);
                  WITH ptr[2]^ DO  BEGIN
                 IF debug THEN  BEGIN
                    writeln('attaching Qgraph, type of node should be 4, is ', ptr[1]^.typeofnode);
                    writeln('number of descendants is ',ptr[1]^.descendants); END;
                 ptr[1]^.descendants := 1 ; {there will be now}
              NEW(ptr[1]^.ptr[1], copnode);
               WITH ptr[1]^.ptr[1]^ DO BEGIN  {with the node we've just made}
                  cop:=XOp;
                  {Edegree :=                 }
                  descendants := 1 ;
                  IF debug THEN BEGIN
                    writeln(' we have created a comp.op node, whose value is ', cop);
                  END;
                  typeofnode := copnode;
                  NEW (ptr[1], valuenode);{now create a Valuenode, pointed to by the copnode}
                  s1 := Svalue; {necessary step due to PRIME bug}
                  ptr[1]^.value := s1;
                  IF debug THEN BEGIN writeln('a valuenode was created, holding the value ', ptr[1]^.value);  END;
                  IF debug THEN writeln( ptr[1]^.value, ' is the leaf.');
                  ptr[1]^.descendants := 0;
                  ptr[1]^.typeofnode := valuenode              END
              END                                           END; {of WITH}

              END
                                   END
                                 END;  { of J-loop }
            midnodestack.nf:=midnodestack.nf-1;
                         END     END           END {of I-loop}
                                                       {WITH}   END;
RF:=RF+1;
END; {of attach}

PROCEDURE processcommand;
{
 PURPOSE:reads relational algebra query in from keyboard

 INPUTS : string from keyboard
 OUTPUTS:if input string is syntactically valid, calls
        appropriate relational algebra procedure
  GLOBALS READ FROM :none
  GLOBALS WRITTEN TO:none
  CALLS          : SQuery
                   get_relation_num
                   get_op_token
                   getname
                   get_att_num
                   project
                   select
```

```
                union
                intersection
                difference
                rename
                join
   IS CALLED BY:main program
   LAST MODIFIED:
   KNOWN BUGS:
   NOTES: this is crude command line parser, for development
          use only. Expects a simple 'three-address' query
          (i.e., of form A <- B binop C or A <- B monop [ ]
           if it discovers an error while processing it
           prints some debugging data and returns to calling
           routine)
}
VAR  S, value,  CStr : STRING[strmax];
          NewR, LeftR, RightR : string[strmax];
          attnames : stcoltype;
          op, ch  : char;
          L    : integer;
          i : integer;
          r1, r2 :relindex_type;
          col_left, col_right : cols_type;
          legalchars : setofchar;
          col : integer;
          compop : coptype;
          comlen:0..strmax;
          oksofar : boolean;
BEGIN

   oksofar := TRUE;
   CStr:='';RightR:='';LeftR:='';NewR:='';s:='';value:='';
   CStr:=SQuery('Enter command line');
   comlen:=length(CStr);
   i:=1 ;
   attnames.count:=0;
   NewR:= getname(i, CStr, ['0'..'9', 'A'..'Z', 'a'..'z'] );
   IF i > comlen THEN BEGIN writeln('i = ', i); oksofar:= FALSE;END;
   WHILE substr(CStr,i,1)=' ' DO i:=i+1;
   i:=i+2;  {to jump over <- }
   LeftR:=getname(i, CStr, ['0'..'9', 'A' .. 'Z', 'a' .. 'z']);
   r1 := get_relation_num(LeftR);
   IF (r1=0) OR (i>comlen)  THEN
          BEGIN writeln('i = ', i);
                writeln('r1 = ', r1);
                oksofar:=FALSE
          END;
   ch:=unstr(substr(CStr,i,1));
   WHILE NOT (ch IN['%',':',';','-','^','+','#','@']) DO
         BEGIN i:=i+1; ch:=unstr(substr(Cstr,i,1)) END;
   IF i>comlen THEN BEGIN writeln('i = ', i); oksofar:=FALSE;END;
   Op := ch;
   IF debug THEN                                          BEGIN
   writeln('command line read was: -->', CStr,'<--');
   writeln('NewR:-->',NewR,'<--');
   writeln('LeftR-->',LeftR,'<--');
   writeln('Op----->',Op,'<--');
   pause;                                                END;
   IF oksofar THEN
   CASE Op OF
     '%', '#': BEGIN
          IF debug THEN writeln('projecting');
          col_left.num:=0;
          WHILE substr(CStr,i,1)<>'[' DO i:=i+1;   i:=i+1;
          WHILE substr(CStr,i,1)<>']' DO BEGIN
             attnames.count:=attnames.count+1;
             s:=getname(i, CStr, ['0'..'9', 'A'..'Z', 'a'..'z']);
             IF debug THEN writeln('attname -->', s, '<--');
             attnames.names[attnames.count]:=s;
             col_left.num := col_left.num + 1;
             col_left.cols[col_left.num]:=get_att_num(s, r1);
             IF col_left.cols[col_left.num]= 0 THEN oksofar:=FALSE;
             IF NOT oksofar THEN
                        writeln('No such attribute as ', s, ' check spelling?');
             WHILE (substr(CStr,i,1)=' ') OR (substr(CStr,i,1)=',') DO i:=i+1;
                                 END;
        IF oksofar THEN  project(NewR, r1,col_left);
       END;
     ':': BEGIN
          IF debug THEN writeln('selecting');
        WHILE substr(Cstr, i, 1) <> '[' DO i:= i+1; i:=i+1;
        s := getname(i, Cstr, ['0'..'9','A'..'Z','a'..'z'] );
        col := get_att_num( s, r1);
        IF debug THEN writeln('The attribute ->',s,'<-',' is column ', col:3);
        s:='';
        s := getname(i, Cstr, ['<','=','>'] );
        compop := get_op_token(s);
        IF debug THEN writeln('Operator and token = ->',s,'<-',compop:2);
        value  := getname(i, Cstr, ['0'..'9', 'A'..'Z', 'a'..'z']);
        IF debug THEN writeln('And the value = ->', value,'<--');
        select(NewR, r1, col, value, compop);
     END;
    '+', '-', '^' : BEGIN
            IF debug THEN writeln('set operation');
            i:=i+1; {skip past op}
            RightR:='';

            RightR := getname(i, Cstr, ['0'..'9', 'A'..'Z', 'a'..'z']);
            IF debug THEN writeln('After getname, RightR ->',RightR,'<--');
            r2:=get_relation_num(RightR);
            IF (r2 < 1) OR (r2 >= rdnf) THEN oksofar := FALSE;
            IF oksofar THEN
              CASE op OF
                '+': union(NewR, r1, r2);
                '-': difference(NewR, r1, r2);
                '^':intersection(NewR, r1, r2);
              END
          END;
    '*': BEGIN
           RightR:=getname(i, Cstr,['0'..'9', 'A'..'Z', 'a'..'z']);
           r2 := get_relation_num(RightR);
           join(NewR, r1, r2,  Eq);
         END;
    '@': BEGIN
         END;
      OTHERWISE writeln(Op,' is an unknown operator.')
    END;     {of CASE Op}
END;
PROCEDURE ireduce( VAR n:nodepointer, op: optype);
{n is a ptr to a relop node called when l and r
 pointers point to identical predications (including
 same sense and permutation}
VAR i : integer;
WITH n^ DO
FOR i := 1 TO ptr[1]^.descendants DO
  attach( op, ptr[1]^.ptr[i]^.ptr[1],
              ptr[2]^.ptr[i]^.ptr[1] );
END;

PROCEDURE Join(          s :STRING; {name of new relation}
                    r1, r2 : INTEGER; {RD location of operands}
                        Op : Coptype {"   <, <=, =,# , >, >=});
                                     {implement for equi-join only}
{Returns new relation header entry in Rdict, plus its derived graph}
VAR  I,J, k : integer;
          c2:integer; {temp fix}

BEGIN   WITH RD[rdnf] DO                                               BEGIN

   Rname := S;
   RDegree := RD[ r1 ].RDegree + RD[ r2 ].RDegree - 1;
   rperm := 1;
   key_cols.num := rdegree ;
   FOR I := 1 TO key_cols.num DO
     key_cols.cols[I] := I;  {mark the key attributes }

   FOR I := 1 TO RD[ r1 ].RDegree DO  WITH attribute[I] DO         BEGIN
        {transfer r1's attribute names}

     Aname:=RD[ r1 ].Attribute[I].Aname;
     Attdomain_Id := RD[ r1 ].attribute[I].Attdomain_ID;           END;


   {now go thru and get all the relevant attribute data from  r2 .
    this will be every attribute except the one being joined over.
    If we later permit multi-column joins this will have to be modified.}

    k:= RD[ r1 ].RDegree + 1;
    FOR I := 1 TO RD[ r2 ].RDegree DO WITH attribute[k] DO          BEGIN
      IF I <> C2 THEN                                               BEGIN
```

```
        Aname := RD[ r2 ].attribute[I].aname;
        Attdomain_ID := RD[ r1 ].attribute[I].Attdomain_ID  END;
      k := k+1                                                    END;
   IF k-1 <> RDegree THEN writeln('error in attribute transfer in JOIN');

   {now set all linked nodes on right = their partners nodes on left}

   Findnodes(RD[rdnf].ptr[1], enode,  midnodestack);
   FOR I:= 1 TO midnodestack.nf-1 DO
     FOR J:= I  TO midnodestack.nf-1 DO
        IF (midnodestack.pointers[I]^.attnum =
            midnodestack.pointers[J]^.attnum)
          AND
            (midnodestack.pointers[I]^.attnum>0)
        THEN
          midnodestack.pointers[J]^.node_id  :=
                midnodestack.pointers[I]^.node_id ;
END                                                    END; {of  join}


PROCEDURE Project( S:STRING; {name of new relation}
                   r1 : INTEGER; {rd location of operand}

              cols_1  : cols_type); {numbers of the columns to be
                                          projected out }

{Returns new relation header entry in rdict, plus its derived graph}

VAR
   Throwout: BOOLEAN; {for deciding to project out a column}
   I, J, k : INTEGER; {loop control and index variables}
   cols_2 : cols_type;
BEGIN WITH rd[rdnf] DO                                              BEGIN
   Rname:=S;
   rdegree := rd[ r1 ].rdegree - cols_1.Num;
{   {new key cols}
   Rperm:= rd[ r1 ].Rperm;
   Preds := rd[ r1 ].preds;
   FOR I:= 1 TO preds DO     ptr[I]:=copy_node(rd[ r1 ].ptr[I]);
   k:=1;
   FOR I:=1 TO rd[ r1 ].rdegree DO                                 BEGIN
      WITH Attribute[k] DO                                   BEGIN
        Throwout:=FALSE;
        FOR J:= 1 TO cols_1.Num DO                           BEGIN
           IF I = cols_1.cols[J] THEN Throwout:=TRUE;
           IF NOT Throwout THEN                         BEGIN
              Aname     := rd[ r1 ].Attribute[I].Aname;
              Attdomain_ID:= rd[ r1 ].Attribute[I].Attdomain_ID;
              k:=k+1;                                     END;
                                                          END   END  END;
     IF debug THEN writeln('k-1 should = rdegree : ', k-1, rdegree);
   { Having copied all the information from the columns we are keeping,
       then go thru and set the attnums
     of the projected out nodes to 0 and reset the attnums of the others}

   cols_2.num := rd[ r1 ].rdegree;
   FOR I:= 1 TO cols_2.num DO  cols_2.cols[i] := maxint;{'unzero' every attribute}

                                          {then zero the projected out ones}
   FOR i:= 1 TO cols_1.num DO cols_2.cols [ cols_1.cols[i] ] := 0;

   j := 1 ;
   FOR i := 1 TO cols_2.num DO{now set the unzeroed ones to the new 1, 2, 3.. values}
     IF cols_2.cols[i] <> 0 THEN    BEGIN
       cols_2.cols[i] := j; j:= j+1   END;

    FOR I:= 1 TO preds DO                                               BEGIN
      ptr[I] := copy_node(rd[ r1 ].ptr[I]);
      midnodestack.nf:=1;
      findnodes(ptr[I], pnode, midnodestack);
      j:= 1;
      WHILE j <  midnodestack.nf DO {for each pnode}                  BEGIN
        midnodestack.pointers[j]^.pperm := map (midnodestack.pointers[j]^ );
        {for each Enode, }
        FOR k:= 1 TO midnodestack.pointers[j]^.descendants DO BEGIN

           midnodestack.pointers[j]^.ptr[k]^.attnum:= cols_2.cols[k] END;
      j:=j+1                                        {WHILE} END
                                                                  {I loop}END
                                                          {of WITH}    END;

rdnf:=rdnf+1;
END; {of project}
PROCEDURE Permute( S:STRING; {name of new relation}
                  R1: INTEGER; {RD location of operand}
             VAR RD: RD_type;
             VAR RF: INTEGER;
                 DD: DD_type;

             Cols_1  : Cols_type); {numbers of the columns to be
                                    permuted, in their new order}

{Returns new relation header entry in Rdict, plus its derived graph}

VAR
   I, J : col_index_type; {loop control and index variables}                    BEGIN
BEGIN WITH RD[RF] DO
   Rname:=S;
   RDegree  := RD[R1].RDegree  ;
   Key_Cols.Num:= RD[R1].Key_Cols.Num;
   FOR I:=1 TO Key_Cols.Num DO
      FOR J:=1  TO Cols_1.Num DO
         IF RD[R1].Key_Cols.Cols[I]=Cols_1.Cols[J]
         THEN Key_Cols.Cols[I]:=J;
         Rperm:= RD[R1].Rperm;{ shall we just change the Rperm?}
   FOR I:=1 TO Rdegree DO                                                  BEGIN
     WITH Attribute[I] DO
       Aname     := RD[R1].Attribute[Cols_1.Cols[I]].Aname;
       Attdomain_ID:= RD[R1].Attribute[Cols_1.Cols[I]].Attdomain_ID  END;

           FOR i := 1 TO preds DO  ptr[i] := copy_node( rd[r1].ptr[i] );

                                                       {of WITH:}   END;

 RF:=RF+1;
 END; {of permute}

 PROCEDURE Union( S:string[strmax]; {name of new relation}
                  R1, R2: INTEGER); {RD location of operand}


 {Returns new relation header entry in Rdict, plus its derived graph}

 VAR
    i, j:INTEGER;
 BEGIN
 RD[rdnf]:=RD[R1];                                                  BEGIN
 WITH  RD[rdnf] DO
    Rname:=S;
 {Now create a new  union node pointed to by ptr[1]
  make its left  descendant a copy of R1's graph,
  make its right descendant a copy of R2's graph.}
        New( ptr[1], Ropnode);
        WITH ptr[1]^ DO                               BEGIN
           typeofnode := Ropnode;
           op := orx;
           descendants := 2;
           ptr[1] := copy_node( RD[r1].ptr[1]); {left}   END;
           ptr[2] := copy_node( RD[r2].ptr[1]); {right}
                                                  {WITH}       END;



   rdnf:=rdnf+1;
  END; {of Union}


  PROCEDURE Difference( S:string[strmax]; {name of new relation}
                        R1, R2: INTEGER); {RD location of operand}


  VAR
     i, j:INTEGER;
  BEGIN
  RD[rdnf]:=RD[R1];
```

```
WITH  RD[rdnf] DO                                                  BEGIN
   Rname:=S;
{Now create a new  difference  node pointed to by ptr[1]
 make its left  descendant a copy of R1's graph,
 make its right descendant a copy of R2's graph.}
      New( ptr[1], Ropnode);
      WITH ptr[1]^ DO                              BEGIN
         typeofnode := Ropnode;
         op := notx;
         descendants := 2;
         ptr[1] := copy_node( RD[r1].ptr[1]); {left}
         ptr[2] := copy_node( RD[r2].ptr[1]); {right}  END;
         {now set all linked nodes on right = their partners nodes on left}

         Findnodes( RD[rdnf].ptr[1] ,enode,midnodestack);
         FOR i:= 1 TO midnodestack.nf-1 DO
            FOR j:= i TO midnodestack.nf-1 DO
               IF (midnodestack.pointers[i]^.attnum =
                     midnodestack.pointers[j]^.attnum)
                  AND
                    (midnodestack.pointers[i]^.attnum>0)
               THEN
                  midnodestack.pointers[j]^.node_id  :=
                  midnodestack.pointers[i]^.node_id ;
                                                      {WITH}       END;



   rdnf:=rdnf+1;
END; {of difference}


PROCEDURE intersection( S:string[strmax]; {name of new relation}
                  R1, R2: INTEGER); {RD location of operand}


{Returns new relation header entry in Rdict, plus its derived graph}

VAR
  i, j, k:INTEGER;
BEGIN
RD[rdnf]:=RD[R1];
WITH  RD[rdnf] DO                                                  BEGIN
 Rname:=S;
 {Now create a new intersect node pointed to by ptr[1]
  make its left  descendant a copy of R1's graph,
  make its right descendant a copy of R2's graph.}
  New( ptr[1], Ropnode);
  WITH ptr[1]^ DO                            BEGIN
   typeofnode := Ropnode;
   op := andx;
   descendants := 2;
   ptr[1] := copy_node( RD[r1].ptr[1]); {left}
   ptr[2] := copy_node( RD[r2].ptr[1]); {right}  END;
   {now set all linked nodes on right = their partners nodes on left}

   Findnodes(RD[rdnf].ptr[1] ,enode,midnodestack);
   FOR i:= 1 TO midnodestack.nf-1 DO
     FOR j:= i TO midnodestack.nf-1 DO
       IF (midnodestack.pointers[i]^.attnum =
           midnodestack.pointers[j]^.attnum)
          AND
          (midnodestack.pointers[i]^.attnum>0)
       THEN                                  BEGIN
          midnodestack.pointers[j]^.node_id  :=
          midnodestack.pointers[i]^.node_id ;

          {transfer Qgraph}
          FOR k:= 1 TO midnodestack.pointers[j]^.descendants DO
           { transfer (  midnodestack.pointers[i]^,
                         midnodestack.pointers[j]^.ptr[k]); }  ;

                                             END
                                            {WITH}       END;
 rdnf:=rdnf+1;
 END; {of intersect}


PROCEDURE Select( S:string; {name of new relation}
                 R1: integer; {RD location of operand}
             Col_num: Col_Index_Type; {Col to be selected on}
             SValue : string; {for selection, value to be compared to}
          '   XOp    : COptype); {"   <, <=, =,# , >, >=}

{Returns new relation header entry in Rdict, plus its derived graph}
        .
VAR
   i, j:integer;
   s1: string[strmax];
   oldgraph: nodepointer;
BEGIN WITH RD[ranf] DO                                                    BEGIN
 Rname:=S;
 RDegree := RD[R1].RDegree; {degree remains the same}
 Rperm:= RD[R1].Rperm; {as does the permutation}
 Key_Cols.Num:=RD[R1].Key_Cols.Num; {nor does the key change}
 FOR i:= 1 TO Key_Cols.Num DO
     Key_Cols.Cols[i]:=RD[R1].Key_Cols.Cols[i];

 FOR i:= 1 TO RDegree DO                              BEGIN
   WITH Attribute[i] DO                       BEGIN
       Aname:=RD[R1].Attribute[i].Aname;
       Attdomain_iD:= RD[R1].Attribute[i].Attdomain_iD;
                                              END     END;

 Preds := RD[R1].Preds;
 FOR i := 1 TO Preds DO
   PTR[i] := copy_node ( RD[R1].Ptr[i] );


 FOR i:= 1 TO preds DO                                                    BEGIN
   midnodestack.nf := 1;
   findnodes(RD[ranf].ptr[i], pnode, midnodestack);
   WITH midnodestack DO                                                 BEGIN
     WHILE nf > 1 DO   BEGIN
       FOR j:= 1 TO pointers[nf-1]^.descendants DO  BEGIN
       {for each of PNode's arguments}
         IF pointers[nf-1]^. ptr[j]^.attnum =col_num THEN  BEGIN

             {attach Qgraph}
            IF pointers[nf-1]^.ptr[j]^.descendants = 0 THEN
            {no graph already attached}
            BEGIN
              pointers[nf-1]^.ptr[j]^.descendants := 1 ;{there will be now}
              NEW(pointers[nf-1]^.ptr[j]^.ptr[1], copnode);
               WITH pointers[nf-1]^.ptr[j]^.ptr[1]^ DO BEGIN
                  cop:=XOp;
                  descendants := 1 ;
                  typeofnode := copnode;
                  NEW (ptr[1], valuenode);
                  s1 := Svalue; {necessary step due to PRIME bug}
                  ptr[1]^.value := s1;
                  ptr[1]^.descendants := 0;
                  ptr[1]^.typeofnode := valuenode              END
             END
             ELSE
             BEGIN   {there is already a QGraph}
           {replace with fully-parameterised insert procedure}
             IF debug THEN BEGIN
                writeln('attaching Qgraph to Enode which has one already');
             END;
               oldgraph := pointers[nf-1]^.ptr[j]^.ptr[1]; {store temporary}
               NEW(pointers[nf-1]^.ptr[j]^.ptr[1], Ropnode);
               WITH pointers[nf-1]^.ptr[j]^.ptr[1]^ DO BEGIN
                  op := andx;
                  typeofnode:=ropnode;
                  descendants := 2;
                  ptr[1] := oldgraph;
                  NEW(ptr[2], copnode);
                  WITH ptr[2]^ DO  BEGIN
                pointers[nf-1]^.ptr[j]^.descendants := 1 ;
              NEW(pointers[nf-1]^.ptr[j]^.ptr[1], copnode);
               WITH pointers[nf-1]^.ptr[j]^.ptr[1]^ DO BEGIN
                  cop:=XOp;
                  {Edegree :=              }
                 descendants := 1 ;
                 typeofnode := copnode;
                 NEW (ptr[1], valuenode);
                 s1 := Svalue; {necessary step due to PRIME bug}
                 ptr[1]^.value := s1;
                 ptr[1]^.descendants := u;
                 ptr[1]^.typeofnode := valuenode              END
            END                                         END; {of WITH}

            END
                                END
                             END;  { of j-loop }
           midnodestack.nf:=midnodestack.nf-1;
                       END    END          END {of i-loop}
                                                     {WITH}   END;

rdnf:=rdnf+1;
END;{ofSelect}

FUNCTION  reduce (   ptr1, ptr2: nodepointer;
                     logicop : roptype;
                  VAR result : result_type
                ): nodepointer ;
{ptr1 and ptr2 are ptrs to Compop nodes
                  each of which points to a valuenode.
                 Must already be in canonical form.}
VAR
  k1, k2 : string[strmax];
       p : nodepointer;
BEGIN {WITH pointer^ DO}
  k1:= ptr1^.ptr[1]^.value; {get left and right leaves}
  k2:= ptr2^.ptr[1]^.value;
  p := NIL ;
  IF k1 < k2 THEN                                                BEGIN
    IF logicop = andx     THEN
      CASE ptr1^.cop OF                            BEGIN
        Stℓ:
                CASE ptr2^.cop   OF
                  stℓ, Seℓ, Neqℓ : result:=left; {reduce to left}
                  Gtℓ, Geℓ,  Eqℓ : result:= empty_set ;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                                   END;
                                                   BEGIN
        Seℓ:
                CASE ptr2^.cop OF
                  Stℓ, Seℓ, Neqℓ : result:=left;;
                  Gtℓ, Geℓ,  Eqℓ : result := empty_set;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                                   END;
                                                   BEGIN
        Neqℓ:
                CASE ptr2^.cop  OF
                  Stℓ, Seℓ, Neqℓ : result:=compose;
                  Gtℓ, Geℓ,  Eqℓ : result:=right;{reduce to right tree}
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                                   END;
                                                   BEGIN
        Eqℓ:
                CASE ptr2^.cop  OF
                  Stℓ, Seℓ, Neqℓ : result:=left;
                  Gtℓ, Geℓ,  Eqℓ :  result := empty_set;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                                   END;
                                                   BEGIN
        Geℓ:
                CASE ptr2^.cop  OF
                  Stℓ, Seℓ, Neqℓ : result:=compose;
                  Gtℓ, Geℓ,  Eqℓ : result:=right;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                                   END;
                                                   BEGIN
        Gtℓ:
                CASE ptr2^.cop  OF
                  Stℓ, Seℓ, Neqℓ :  result:=compose;
                  Gtℓ, Geℓ,  Eqℓ : result:=right;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Ssf:
              CASE ptr2^.cop  OF
              Stf, Sef, Neqf,
                Gtf, Gef,  Eqf  : result := incompatible_operators;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       otherwise                                END

       END (of CASE) ;
                                                    END
                                                          BEGIN
  ELSE (op = ox, SO )
    CASE ptr1^.cop OF
                                                BEGIN
       Stf:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  : result::=right;
                Gtf, Gef,  Eqf  : result::=compose;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Sef:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  : result::=right;
                Gtf, Gef,  Eqf  : result::=compose;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Neqf:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  : result := universal_set;
                Gtf, Gef,  Eqf  : result:= left;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Eqf:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  : result::=right;
                Gtf, Gef,  Eqf  : result::=compose   ;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Gef:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  :result::=compose;
                Gtf, Gef,  Eqf  : result:= left;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Gtf:
              CASE ptr2^.cop  OF
                Stf, Sef, Neqf  :result::=compose;
                Gtf, Gef,  Eqf  : result::=left;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       Ssf:
              CASE ptr2^.cop OF
                Stf, Sef, Neqf  ,
                Gtf, Gef,  Eqf  : result := incompatible_operators;
                Ssf             : BEGIN END;
                otherwise         BEGIN writeln('error in P_reduce')END
              END (of CASE);
                                                END;
                                                BEGIN
       otherwise                                END

     END (of CASE) ;
                                                    END
  ELSE ( k1 = k2 )                                              BEGIN
    IF logicop = anox  THEN
```

```
CASE ptr1^.cop OF
  Stℓ:                                         BEGIN
          CASE ptr2^.cop OF
             Stℓ: result:=either;
             Seℓ, Neqℓ : result:=left;;
            Gtℓ, Geℓ,  Eqℓ : result := empty_set;
            Ssℓ               : BEGIN END;
    ;       otherwise           BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                               END;
  Seℓ:                                         BEGIN
  CASE ptr2^.cop  OF
            Stℓ :result:=right;
            Seℓ: result:=either;
            Neqℓ           :BEGIN
                 ptr1^.cop:=Stℓ;
                    result:=newop;
                               END;
             Eqℓ              : result:=right;
            Geℓ               :BEGIN
                  p:=copy_node(ptr1);     p^.cop:=Eqℓ;
                 result:=newop;
                                 END;
            Gtℓ               : result := empty_set;
            Ssℓ               : BEGIN END;
            otherwise           BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                               END;
  Neqℓ:                                        BEGIN
          CASE ptr2^.cop  OF
             Stℓ,  Neqℓ :result:=right;
             Seℓ:          BEGIN
                  p:=copy_node(ptr1);     p^.cop:=Stℓ;
                 result:=newop;
                                               END;
                 Eqℓ : result := empty_set;
                 Geℓ: BEGIN
                  p:=copy_node(ptr1);     p^.cop:=Gtℓ;
                         END;
                 Gtℓ: result:=right;
             Ssℓ              : BEGIN END;
             otherwise          BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                               END;
  Eqℓ:                                         BEGIN
          CASE ptr2^.cop  OF
             Stℓ: result := empty_set;
             Seℓ: result:=left;
           Neqℓ: result := empty_set;
              Eqℓ :  result := either;
                    Geℓ : result := left;
            Gtℓ :          result := empty_set;
            Ssℓ               : BEGIN END;
            otherwise           BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                               END;
  Geℓ:                                         BEGIN
          CASE ptr2^.cop  OF
             Stℓ:     result := empty_set;
             Seℓ:     BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Eqℓ;
                     result:=newop;
                      END;
             Neqℓ:     BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Gtℓ;
                     result:=newop;
                      END;
             Gtℓ,   Eqℓ :      result:=right ;
                  Geℓ : result:=either;
             Ssℓ              : BEGIN END;
             otherwise          BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                               END;
  Gtℓ:                                         BEGIN
          CASE ptr2^.cop  OF
             Stℓ, Seℓ    :      result := empty_set;
             Eqℓ             : result := empty_set;
              Geℓ,   Neqℓ : result:=left;
                 Gtℓ : result:=either;
             Ssℓ                : BEGIN END;
            otherwise          BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     Ssℓ:                                   BEGIN
          CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ ,
            Gtℓ, Geℓ,  Eqℓ : result := incompatible_operators;
     :      Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     otherwise                              BEGIN
                                            END
     END (of CASE) ;

                                                END         BEGIN
ELSE (op = orx, SO )
     CASE ptrl^.cop OF
     Stℓ:                                   BEGIN
          CASE ptr2^.cop  OF
             Stℓ : result:=either;
            Seℓ, Neqℓ : result:=right ;
            Eqℓ: BEGIN
                     p:=copy_node(ptrl);    p^.cop:=Seℓ;
                    result:=newop;
                   END;
            Geℓ: result := universal_set;
            Gtℓ: BEGIN
                     p:=copy_node(ptrl);    p^.cop:=Neqℓ;
                    result:=newop;
                   END;
            Ssℓ             : BEGIN END;
            otherwise         BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     Seℓ:                                   BEGIN
          CASE ptr2^.cop  OF
            Stℓ,  Eqℓ : result:=left;
              Seℓ  : result := either;
            Gtℓ, Geℓ,  Neqℓ : result := universal_set;
            Ssℓ             : BEGIN END;
            otherwise         BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     Neqℓ:                                  BEGIN
          CASE ptr2^.cop  OF
            Stℓ, Gtℓ : result:= left;
                 Neqℓ : result:= either;
            Seℓ, Geℓ,  Eqℓ : result := universal_set;
            Ssℓ             : BEGIN END;
            otherwise         BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     Eqℓ:                                   BEGIN
          CASE ptr2^.cop  OF
            Stℓ: BEGIN
                    result:=newop;
                     p:=copy_node(ptrl);    p^.cop:=Seℓ;
                   END;
           Neqℓ: result:= universal_set;
            Seℓ,  Geℓ: result:=right;
                Eqℓ : result:= either;
            Gtℓ: BEGIN
                     p:=copy_node(ptrl);    p^.cop:=Geℓ;
                    result:=newop;
                   END;
            Ssℓ             : BEGIN END;
            otherwise         BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                            END;
     Geℓ:                                   BEGIN
          CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ : result := universal_set;
            Gtℓ,   Eqℓ : result:=left;

              Geℓ: result:= either;
            Ssℓ             : BEGIN END;
            otherwise         BEGIN writeln('error in p_reduce')END
          END (of CASE);
                                            END;
     Gt£:                                        BEGIN
               CASE ptr2^.cop OF
                 St£:BEGIN
                        p:=copy_node(ptr1);     p^.cop:=Neq£;
                       result:=newop;
                    END;
                 Se£: result := universal_set;
                Neq£, Ge£: result:=right;
                 Eq£:  BEGIN
                       result:=newop;
                        p:=copy_node(ptr1);     p^.cop:=Ge£;
                      END;
                 Gt£:result:=either;
                 Ss£              : BEGIN END;
                 otherwise          BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                                 END;
     Ss£:                                        BEGIN
               CASE ptr2^.cop  OF
                 St£, Se£, Neq£ ,
                 Gt£, Ge£,  Eq£  : result := incompatible_operators;
                 Ss£              : BEGIN END;
                 otherwise          BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                                 END;
     otherwise                                   BEGIN
                                                 END
 END (of CASE) ;
                                                     END;
END (of CASE k1 = k2 );

CASE result OF
    empty_set, universal_set: p:=NIL;
  incompatible_operators  :
        writeln('Error -- can''t mix operators');
    left    :  p:=copy_node(ptr1);
   right    :  p:=copy_node(ptr2);
  either    :  p:=copy_node(ptr1);
  compose   :                                   BEGIN
               {new( pointer, ropnode);
               WITH pointer^ DO       BEGIN
                 typeofnode:=ropnode;
                 descendants:=2;
                 op:=logicop;
                 ptr[1]:=copy_node(ptr1);
                 ptr[2]:=copy_node(ptr2)    END; p:=pointer^ }     END;
        newop: { action already accomplished }
 END (of CASE);

 reduce:=p
END (of reduce);


PROCEDURE utter(N:Nodepointer; R: RelIndex_type; D: DomIndex_type);
CONST  NULL = '*';(token for 'say nothing' in predication phrases)
      inc = 2;
VAR
   i                         : 0..3; (descendants counter)
   similar_count, same_count : 0..maxint;
   next : 1..stackmax;
   Entity, Phrase : 1..24;
   SEname, Rel_pro, cophrase,rep_name_holder,
   PEname, Indicator, Edegphrs, Prdphrs , atnam : String[strmax];
   case£: case_type;

PROCEDURE indent(n:integer);
BEGIN IF n>0 THEN write(' ':n) END;

PROCEDURE report;
BEGIN
   writeln('gsuppress = ', gsuppress, 'gsense = ',gsense);
   writeln('glevel    = ', glevel   , 'gprojected_out = ', gprojected_out);
END;


BEGIN WITH  N^ DO                                   BEGIN

   IF debug THEN writeln('utter called, typeofnode = ', typeofnode);
   CASE  Typeofnode    OF

     Ropnode:                                            BEGIN
       glevel:=glevel + inc;
{       indent(glevel);}
       IF Op=Orx THEN writeln('(either ') ;

       utter ( Ptr[1], R, D );

       glevel:=glevel-inc;
{       indent(glevel);}
       CASE  Op  OF
         Orx  : writeln ('or ');
         Andx: IF ptr[2]^.typeofnode = pnode THEN
                  IF ptr[2]^.psense = negative THEN
                     writeln( 'but')
                  ELSE writeln('and')
                ELSE writeln('and');
         Notx : BEGIN Writeln ('but it is NOT the case that  ');
                      gsense := negative
                END;
       END; {of CASE}

       utter( ptr[2], R, D);
       glevel:=glevel-inc;
       gsense:=positive;
       IF op=Orx THEN writeln('or both)') ;
                                                                     END;{of Rop}

     Copnode:                                                  BEGIN

        glevel:=glevel+inc;
            {Now decide upon the actual phrase to use for the
             Comparison operation. 'Ordered' means 'does this
             domain use the default phrases ('less than, equal to',
             etc.) or does it use some domain-specific phraseology
             supplied by the user, eg 'is' or 'brighter than'.
             Ordered = No -> default values,
             Ordered = Yes -> get user-supplied values from DD )


       IF gprojected_out AND (cop=Neq£) THEN cophrase := 'other than ' ELSE
       IF gprojected_out AND (cop=eq£)  THEN cophrase := 'including '
       ELSE              cophrase :=  DD[D].OrderData.CompPhrases[Cop];

       IF (NOT (gsuppress)) THEN write(cophrase,' ');

       utter( Ptr[1], R, D );{should point to a Valuenode}
       glevel:=glevel-inc;
                                                                  END;{of Copnode}

     Pnode:                                                          BEGIN

{       glevel:=glevel+inc; }
{       indent(glevel);}
    FOR Phrase := 1 TO Descendants DO                  BEGIN
      Entity:=Calculate_Order (Pperm, Phrase, P_Table) ;

      utter(ptr[Entity], R, D);


      IF Phrase <  Descendants THEN                BEGIN

       IF Alt_sentences[Pperm, phrase].Main = NO THEN
          Prdphrs := Alt_sentences[Pperm, phrase].Case_phrase
       ELSE
          Prdphrs := Alt_sentences[Pperm, phrase].Alt_phrases[Psense, Pnumber];
       IF Prdphrs = NULL THEN Prdphrs:=' ';
         Writeln(Prdphrs,' ')   END END {of phrase-loop,  and   Pnode: } END;


   Enode:                                                             BEGIN
       IF debug THEN writeln('Entering Enode.');
       IF debug THEN report;
         gsuppress:=FALSE;
         gprojected_out := attnum=0 ;
         IF   (DD[entdomain_id].self_id=YES) AND (descendants>0) THEN
                IF ptr[1]^.typeofnode= Copnode THEN
                   IF ptr[1]^.cop = Eq£ THEN
                gsuppress := TRUE;
 {see if this entity has been uttered before}
same_count:=0; similar_count:=0;


next := midnodestack.nf-1;
WHILE next > 1 DO                                        BEGIN
  IF midnodestack.pointers[next]^.node_id = node_id
  THEN same_count:= same_count +1
  ELSE
    IF midnodestack.pointers[next]^.Entdomain_id = Entdomain_id
    THEN similar_count := similar_count + 1;
  next := next -1                                          END;


 {get Indicator phrase for this entity}

IF gsense = positive THEN
  IF NOT(gprojected_out) THEN
    IF same_count = 0 THEN {it's a Principal Enode}
      IF descendants > 0 THEN  Indicator:='the indicated '
      ELSE                     Indicator:='the indicated '
    ELSE {same_count > 0, so it's not a Principal Enode}
      IF descendants > 0 THEN  Indicator:='error 1'
      ELSE                     Indicator:='the indicated '
  ELSE {attnum = 0}
    IF same_count = 0 THEN
      IF descendants > 0 THEN
        IF Edegree = one THEN  Indicator := 'a single '
        ELSE                   Indicator := 'at least one '
      ELSE
        IF Edegree = one THEN  Indicator := 'a single '
        ELSE                   Indicator := 'at least one '
    ELSE {same_count > 0, it's not a Principal Enode}
      IF descendants > 0 THEN
        IF Edegree = one   THEN Indicator := 'a single '
        ELSE                    Indicator := 'at least one '
      ELSE                      Indicator := 'at least one '
ELSE { gsense = negative, SO}
  IF NOT(gprojected_out) THEN
    IF same_count = 0 THEN {it's a Principal Enode}
      IF descendants > 0 THEN  Indicator:='error 2 '
      ELSE                     Indicator:='the indicated '
    ELSE {same_count > 0, so it's not a Principal Enode}
      IF descendants > 0 THEN  Indicator:='an indicated'
      ELSE                     Indicator:='the indicated '
  ELSE {attnum = 0}
    IF same_count = 0 THEN
      IF descendants > 0 THEN
        IF Edegree = one THEN  Indicator := 'any '
        ELSE                   Indicator := 'at least one '
      ELSE
        IF Edegree = one THEN  Indicator := 'any '
        ELSE                   Indicator := 'any '
    ELSE {same_count > 0, it's not a Principal Enode}
      IF descendants > 0 THEN
        IF Edegree = one   THEN Indicator := 'a single '
        ELSE                    Indicator := 'those '
      ELSE                      Indicator := 'that ';

 IF Indicator='' THEN Indicator:='ERROR -- No indic.phrase assigned';
 IF debug THEN writeln('preparing to write Indicator');
 IF NOT(gsuppress) THEN write(Indicator);


 SEname :=DD[Entdomain_id].entity_set_name[singular];
 PEname:=DD[Entdomain_id].entity_set_name[plural];

 IF role[singular]<>'none' THEN                                BEGIN
    SEname := SEname + ' (acting as a ' + role [singular] +' )';
    PEname :=PEname + ' (acting as '   + role [plural   ] +' )'  END;

 IF NOT(gsuppress) THEN write(SEname,' ');

 IF (NOT(gprojected_out)) AND (NOT(gsuppress)) THEN  BEGIN
    atnam := RD[R].attribute[attnum].aname;
    write('[', atnam, '] ')                          END;
```

```
      {here we generate any Qualification Graphs, such as
        'which is greater than 5,000 and less than 10,000 '    or
        'which is French'
        'which are other than French'  }


        IF DD[D].repdata.represented = YES THEN  BEGIN
                rel_pro:= DD[D].repdata.repname[singular];
                rel_pro:= 'whose ' + rel_pro           END
        ELSE
               rel_pro:=get_rel_pro(nom,D);

     glevel:=glevel+1;
{     indent(glevel);}
    IF (descendants > 0) AND (NOT(gsuppress)) THEN write(rel_pro, ' ');

     FOR i:= 1 TO descendants DO                         BEGIN
      IF debug THEN BEGIN writeln('uttering descendants.')
      END;
       utter ( Ptr[i], R, Entdomain_id )                  END;


        IF gsense = positive  THEN
          CASE EDegree OF
            exported :  BEGIN
                {here we do nothing since we have already put the Edegphrs
                          information earlier, after the qualifying graph
                          for this entity}
                      END;
            one :     BEGIN
                        Edegphrs:='(and this  '+ SEname + ' only) ';
                        {Edegphrs := '(and no other ' + PEname + ')';}
                      END;
            many :    BEGIN
                       Edegphrs:='(and possibly other ' + PEname+') ';
                      END;
             all :     BEGIN {possible future development}
                        Edegphrs:='(and all other ' + PEname+ ' )';
                       END;
          END  {of CASE}
      ELSE {gsense = negative, SO}                              BEGIN
            Edegphrs:=' ';                                        END;
       writeln(Edegphrs,' ')              { Edegphrs block : }     ;


     push(N, midnodestack);                                          END;


      Valuenode:  writeln( Value ,' ');


END;{of global case}
END;{of WITH}
END;{of UTTER}

FUNCTION  reduce (    ptr1, ptr2: nodepointer;
                      logicop : roptype;
                  VAR result : result_type
                ): nodepointer ;
{ptr1 and ptr2 are ptrs to Compop nodes
                  each of which points to a valuenode.
                  Must already be in canonical form.}

VAR
  k1, k2 : string[strmax];
       p : nodepointer;
BEGIN {WITH pointer^ DO}
  k1:= ptr1^.ptr[1]^.value; {get left and right leaves}
  k2:= ptr2^.ptr[1]^.value;
  p := NIL ;
  IF k1 < k2 THEN
    IF logicop = andx    THEN                                    BEGIN
      CASE ptr1^.cop OF
        Stl:                                   BEGIN
                 CASE ptr2^.cop   OF
                   Stl, Sel, Neql : result:=left; {reduce to left}
                   Gtl, Gel,  Eql : result:= empty_set ;
                   Ssl            : BEGIN END;
                   otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
                                        BEGIN
   Seℓ:   CASE ptr2^.cop OF
            Stℓ, Seℓ, Neqℓ : result:=left;;
            Gtℓ, Geℓ,  Eqℓ : result := empty_set;
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
                                        BEGIN
   Neqℓ:  CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ : result:=compose;
            Gtℓ, Geℓ,  Eqℓ : result:=right;(reduce to right tree)
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASL);
                                        END;
                                        BEGIN
   Eqℓ:   CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ : result:=left;
            Gtℓ, Geℓ,  Eqℓ :  result := empty_set;
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
                                        BEGIN
   Geℓ:   CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ : result:=compose;
            Gtℓ, Geℓ,  Eqℓ : result:=right;
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
                                        BEGIN
   Gtℓ:   CASE ptr2^.cop  OF
            Stℓ, Seℓ, Neqℓ :  result:=compose;
            Gtℓ, Geℓ,  Eqℓ : result:=right;
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
                                        BEGIN
   Ssℓ:   CASE ptr2^.cop  OF
          Stℓ, Seℓ, Neqℓ,
            Gtℓ, Geℓ,  Eqℓ : result := incompatible_operators;
            Ssℓ            : BEGIN END;
            otherwise        BEGIN writeln('error in P_reduce')END
          END (of CASE);
                                        END;
   otherwise                            BEGIN
                                        END
   END (of CASE) ;
                                                END
                                                      BEGIN
ELSE (op = ox, SO )
  CASE ptr1^.cop OF
                                        BEGIN
    Stℓ:    CASE ptr2^.cop  OF
              Stℓ, Seℓ, Neqℓ : result:=right;
              Gtℓ, Geℓ,  Eqℓ : result:=compose;
              Ssℓ            : BEGIN END;
              otherwise        BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                        END;
                                        BEGIN
    Seℓ:    CASE ptr2^.cop  OF
              Stℓ, Seℓ, Neqℓ : result:=right;
              Gtℓ, Geℓ,  Eqℓ : result:=compose;
              Ssℓ            : BEGIN END;
              otherwise        BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                        END;
                                        BEGIN
    Neqℓ:   CASE ptr2^.cop  OF
              Stℓ, Seℓ, Neqℓ : result := universal_set;
              Gtℓ, Geℓ,  Eqℓ : result:= left;
              Ssℓ            : BEGIN END;
              otherwise        BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                        END;
```

```
      Eqℓ:                                      BEGIN
               CASE ptr2^.cop  OF
                 Stℓ, Seℓ, Neqℓ : result:=right;
                 Gtℓ, Geℓ,  Eqℓ : result:=compose  ;
                 Ssℓ            : BEGIN END;
                 otherwise        BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                              END;
      Geℓ:                                    BEGIN
               CASE ptr2^.cop  OF
                 Stℓ, Seℓ, Neqℓ :result:=compose;
                 Gtℓ, Geℓ,  Eqℓ : result:= left;
                 Ssℓ            : BEGIN END;
                 otherwise        BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                              END;
      Gtℓ:                                    BEGIN
               CASE ptr2^.cop  OF
                 Stℓ, Seℓ, Neqℓ :result:=compose;
                 Gtℓ, Geℓ,  Eqℓ : result:=left;
                 Ssℓ            : BEGIN END;
                 otherwise        BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                              END;
      Ssℓ:                                    BEGIN
               CASE ptr2^.cop OF
                 Stℓ, Seℓ, Neqℓ ,
                 Gtℓ, Geℓ,  Eqℓ : result := incompatible_operators;
                 Ssℓ            : BEGIN END;
                 otherwise        BEGIN writeln('error in P_reduce')END
               END (of CASE);
                                              END;
      otherwise                               BEGIN
                                              END
  END (of CASE) ;
                                                   END
ELSE ( k1 = k2 )                                               BEGIN
  IF logicop = andx  THEN
    CASE ptr1^.cop OF
      Stℓ:                                     BEGIN
                CASE ptr2^.cop OF
                  Stℓ: result:=either;
                  Seℓ, Neqℓ : result:=left;;
                 Gtℓ, Geℓ,  Eqℓ : result := empty_set;
                 Ssℓ            : BEGIN END;
                 otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                               END;
      Seℓ:                                     BEGIN
      CASE ptr2^.cop  OF
                  Stℓ :result:=right;
                  Seℓ: result:=either;
                  Neqℓ          :BEGIN
                       ptr1^.cop:=Stℓ;
                         result:=newop;
                                END;
                   Eqℓ           : result:=right;
                  Geℓ            :BEGIN
                          p:=copy_node(ptr1);    p^.cop:=Eqℓ;
                         result:=newop;
                                  END;
                  Gtℓ            : result := empty_set;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
                END (of CASE);
                                               END;
       Neqℓ:                                   BEGIN
                CASE ptr2^.cop  OF
                  Stℓ,  Neqℓ :result:=right;
                  Seℓ:           BEGIN
                          p:=copy_node(ptr1);    p^.cop:=Stℓ;
                         result:=newop;
                                               END;
                         Eqℓ : result := empty_set;
                         Geℓ: BEGIN
                          p:=copy_node(ptr1);    p^.cop:=Gtℓ;
                             END;
                         Gtℓ: result:=right;
                  Ssℓ            : BEGIN END;
                  otherwise        BEGIN writeln('error in P_reduce')END
```

```
            END (of CASE);
                                                END;
    Eqℓ:                                        BEGIN
            CASE ptr2^.cop  OF
              Stℓ: result := empty_set;
              Seℓ: result:=Left;
             Neqℓ: result := empty_set;
                Eqℓ :  result := either;
                     Geℓ : result := left;
              Gtℓ :          result := empty_set;
              Ssℓ                 : BEGIN END;
              otherwise             BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                                END;
    Geℓ:                                        BEGIN
            CASE ptr2^.cop  OF
              Stℓ:    result := empty_set;
              Seℓ:    BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Eqℓ;
                     result:=newop;
                      END;
              Neqℓ:    BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Gtℓ;
                     result:=newop;
                      END;
              Gtℓ,    Eqℓ :     result:=right ;
                 Geℓ : result:=either;
              Ssℓ                 : BEGIN END;
              otherwise             BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                                END;
    Gtℓ:                                        BEGIN
            CASE ptr2^.cop  OF
              Stℓ, Seℓ     :   result := empty_set;
              Eqℓ          : result := empty_set;
               Geℓ,   Neqℓ : result:=left;
                  Gtℓ : result:=either;
              Ssℓ                 : BEGIN END;
              otherwise             BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                                END;
    Ssℓ:                                        BEGIN
            CASE ptr2^.cop  OF
              Stℓ, Seℓ, Neqℓ :
              Gtℓ, Geℓ,  Eqℓ : result := incompatible_operators;
              Ssℓ            : BEGIN END;
              otherwise        BEGIN writeln('error in P_reduce')END
            END (of CASE);
                                                END;
    otherwise                                   BEGIN
                                                END
    END (of CASE) ;

                                                   END        BEGIN
ELSE (op = orx, So )
    CASE ptr1^.cop OF                           BEGIN
    Stℓ:
             CASE ptr2^.cop  OF
                Stℓ : result:=either;
               Seℓ, Neqℓ : result:=right ;
               Eqℓ: BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Seℓ;
                     result:=newop;
                    END;
               Geℓ: result := universal_set;
               Gtℓ: BEGIN
                      p:=copy_node(ptr1);     p^.cop:=Neqℓ;
                     result:=newop;
                    END;
               Ssℓ               : BEGIN END;
               otherwise           BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                END;
    Seℓ:                                        BEGIN
             CASE ptr2^.cop  OF
               Stℓ,  Eqℓ : result:=left;
                 Seℓ  : result := either;
               Gtℓ, Geℓ,  Neqℓ : result := universal_set;
               Ssℓ               : BEGIN END;
               otherwise           BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                        END;
       Neqℓ:                                            BEGIN
             CASE ptr2^.cop  OF
               Stℓ, Gtℓ : result:= left;
                      Neqℓ : result:= either;
               Seℓ, Geℓ,  Eqℓ : result := universal_set;
               Ssℓ              : BEGIN END;
               otherwise          BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                        END;
                                                        BEGIN
       Eqℓ:
             CASE ptr2^.cop  OF
               Stℓ: BEGIN
                        result:=newop;
                         p:=copy_node(ptr1);     p^.cop:=Seℓ;
                    END;
              Neqℓ: result:= universal_set;
               Seℓ,  Geℓ: result:=right;
                    Eqℓ : result:= either;
               Gtℓ: BEGIN
                         p:=copy_node(ptr1);     p^.cop:=Geℓ;
                        result:=newop;
                    END;
               Ssℓ              : BEGIN END;
               otherwise          BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                        END;
                                                        BEGIN
       Geℓ:
             CASE ptr2^.cop  OF
               Stℓ, Seℓ, Neqℓ : result := universal_set;
               Gtℓ,   Eqℓ : result:=left;

                 Geℓ: result:= either;
               Ssℓ              : BEGIN END;
               otherwise          BEGIN writeln('error in p_reduce')END
             END (of CASE);
                                                        END;
                                                        BEGIN
       Gtℓ:
             CASE ptr2^.cop OF
               Stℓ:BEGIN
                         p:=copy_node(ptr1);     p^.cop:=Neqℓ;
                        result:=newop;
                    END;
               Seℓ: result := universal_set;
              Neqℓ, Geℓ: result:=right;
               Eqℓ:  BEGIN
                        result:=newop;
                         p:=copy_node(ptr1);     p^.cop:=Geℓ;
                       END;
               Gtℓ:result:=either;
               Ssℓ              : BEGIN END;
               otherwise          BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                        END;
                                                        BEGIN
       Ssℓ:
             CASE ptr2^.cop  OF
               Stℓ, Seℓ, Neqℓ :
               Gtℓ, Geℓ,  Eqℓ : result := incompatible_operators;
               Ssℓ              : BEGIN END;
               otherwise          BEGIN writeln('error in P_reduce')END
             END (of CASE);
                                                        END;
        otherwise                                       BEGIN
                                                        END
   END (of CASE) ;
                                                              END;
  END (of CASE k1 = k2 );

  CASE result OF
      empty_set, universal_set: p:=NIL;
    incompatible_operators   :
         writeln('Error -- can''t mix operators');
      left     :  p:=copy_node(ptr1);
     right     :  p:=copy_node(ptr2);
    either     :  p:=copy_node(ptr1);
    compose    :                                        BEGIN
                   (new( pointer, ropnode);
                   WITH pointer^ DO        BEGIN
```

```
                    typeofnode:=ropnode;
                    descendants:=2;
                    op:=logicop;
                    ptr[1]:=copy_node(ptr1);
                    ptr[2]:=copy_node(ptr2)   END; p:=pointer^ )     END;
             newop: { action already accomplished }
 END (of CASE);


 reduce:=p
END (of reduce);
which
which
who
whom

{ $E+}
CONST
  maxreal =  999.0; {change this later}
 menumax    = 20;{used in menu routines -- max choices permitted}
 stackmax = 24;
 maxvalues = 16;
 degmax     = 12;{max degree of any relation}
 attmax     =  6;
 maxpreds   = 12; {max number of predications a relation can have}
 permmax    =  6; {Must always be  factorial of largest base relation}
                   {degree permitted}
 dmax       =  6; {max number of entity/domains allowed in data dictionary }
 rmax       = 16; {max number of relations -- base and derived -- allowed}

                  {The following are synonyms used in type definitions
                   They are the equivalent of user-defined enumerated
                   types. However, unlike enumerated types, they can
                   be input and output to files/the screen, using
                   standard read and write procedures, which saves writing a
                   special I/O procedure for each type (at the expense
                   of a certain amount of built-in error checking)    }

 no             = 0 ;
 yes            = 1 ;

 negative       = 0 ;
 positive       = 1 ;

 fattribute     = 0;
 relationship   = 1;

 singular       = 0 ;
 plural         = 1 ;

 exported       = 0;
 one            = 1;
 many           = 2;
 all            = 3;

 ent_att        = 0;
 ent_ent        = 1;

 nom            = 0;
 obj            = 1;

 inanimate      = 0;
 animate        = 1;

 Bool           = 0;
 int            = 1;
 rel            = 2;
 Cha            = 3;
 Str            = 4;
 dat            = 5;   {for future expansion}

 ropnode        = 1;
 copnode        = 2;
 pnode          = 3;
 enode          = 4;
 valuenode      = 5;

 orx            = 0;
 andx           = 1;
 notx           = 2;
```

```
Neqf          = 1;
Eqf           = 2;
Stf           = 3;
Sef           = 4;
Gef           = 5;
Gtf           = 6;
Ssf           = 7;

{{P}
TYPE
    result_type = (empty_set, universal_set, left, right, either, compose, newop, incompatible_operators);
   File_name_type = ARRAY [ 1..32] OF CHAR;
   StColType = RECORD
                   count: 0..24;
                   names: ARRAY[1..24] OF string[80];
                 END;
   CharArraytype = ARRAY[1..40] OF char;
   animate_type = inanimate..animate;
   number_type = singular..plural;
   case_type    = nom..obj;

   Domindex_type  = 1..dmax;
   relindex_type  = 1..rmax;
   col_index_type = 1..degmax;
   cols_type      = RECORD
                      Num: col_index_type;
                      cols: ARRAY[col_index_type] OF integer;
                    END;
  Visit_count_type = 0 .. 12;

  att_index_type    = 1..attmax;
  perm_type         = 1..Permmax;
  Edegree_type      = exported .. all;

  menu_Choice_type = 0..menumax;
  menu_type = RECORD
                Choices : menu_Choice_type;
                Options : ARRAY [ menu_Choice_type] OF string[80];
              END;

  p_table_type  =  ARRAY [ 1..24, 1..4 ] OF  integer;
  pro_table_type = ARRAY [ inanimate..animate, nom..Obj] OF string[80];

  yesno_type  = no .. yes ;

{Here are all the type definitions relating to the RT apparatus}


  nodepointer = ^node;
  node_id_type = 0..maxint;

  pointerstack = RECORD
                    nf : 1..stackmax;
                    pointers : ARRAY[ 1.. stackmax] OF nodepointer;
                 END;
  nodetype =  ropnode .. valuenode;
  setofnodetypes = SET OF nodetype;
  setofchar       = SET OF char;

  roptype = orx .. notx ;
  coptype = Neqf .. Ssf ;


  Sense_type  = negative .. positive;


  phrase_type =    RECORD
                     Main: yesno_type;
                     CASE Main:yesno_type  OF
                             no:  ( Case_phrase  : String[80]);
                             yes: ( Alt_phrases  : ARRAY[Sense_type, number_type]
                                        OF String[80]);
                   END;

  Sentence_type    = ARRAY [ 1..degmax] OF phrase_type;

  node  = RECORD
           descendants : 1..degmax; {number of pointers not NIL}
           ptr         : ARRAY [1..degmax] OF nodepointer;
```

```
      typeofnode  : nodetype;

      CASE typeofnode:nodetype OF
        ropnode : (Op:roptype);
        copnode : (cop:coptype;
                   cdegree: Edegree_type; )
        pnode   : (Pred_id : 1..rmax; {unique predication identifier}
                   pperm    : Perm_type ; {which permutation it is}
                   pred_type : ent_att..ent_ent; {some predications of
                                                  degree two are entity/
                                                  attribute ones}

                   psense  : Sense_type;
                   pnumber : singular .. plural;
                   Alt_sentences  : ARRAY [perm_type] OF Sentence_type);
           enode   : (
                 attnum       : 0 .. degmax; { attribute  entity refers to}
                 Entdomain_id : domindex_type; {domain of this argument}
                 dummy        : yesno_type;
                 edegree      : edegree_type;
                 node_id      : node_id_type;
                 role         : ARRAY [ Number_type] OF  string[80];
                 trimmed      : yesno_type;
                 shortdesc    : string[80]);
        valuenode: (value:string[80]);
      END; {of 'node'}


{These are the data Dictionary types which hold information about
 each relation -- both Base relations and derived relations -- and
 each domain   -- note that Domain is synonymous with entity as far
 as reverse translation is concerned.}

attHdrtype = RECORD
               aname   : string[80];
               Attdomain_id : 1..dmax; {location  of  domain in  data dict}
             END;

rhdrtype = RECORD
             Rname      : string[80];
             Rdegree    : 1..24;
             Key_cols   : cols_type;
             Rperm      : Perm_type;
             attribute  : ARRAY[1..24] OF attHdrtype;
             preds      : 1..maxPreds; {every relation has at least one}
             ptr        : ARRAY [1..maxPreds] OF nodepointer;
           END;

DomDictype =
  RECORD
     entity_set_name    : ARRAY [ number_type ] OF string[80];

     animate_value  : inanimate .. animate;
     Self_ID        : yesno_type;

     repdata : RECORD
        represented : yesno_type;
        CASE represented : yesno_type  OF
             no: ( );
             yes: (
                   repName : ARRAY [ singular  .. plural]   OF string[80]);
       END; {of repdata record}

     Rangedata : RECORD
       datatype : Bool .. dat;
       CASE datatype: Bool .. dat  OF
            Bool, Cha, Str  : ( );
            rel             : (rmin, rmax : REAL);
            int             :(imin, imax : integer );
       END;{of Rangedata record}

     orderdata : RECORD
       ordered : yesno_type;
       compphrases : ARRAY [Neqf..Ssf] OF string[80];
      END; {of orderdata}

     valuesetdata : RECORD
        Limited : yesno_type;
          CASE Limited: yesno_type  OF
            no: ( );
          yes: ( values : RECORD
                          count :       1..maxvalues;
                        Stringℓ : ARRAY  [ 1..maxvalues]
                                   OF string [80] ;
                              END; (of values record));
     END; (of valuesetdata)

END;( of RECORD)

ra_type = ARRAY [1..rmax] OF rhdrtype;
dd_type = ARRAY [1..dmax] OF DomDictype;

(end of TYPE definitions)

VAR
  gsense : sense_type; (for controlling determinant phrases)
  gsuppress: boolean; (for suppressing utterance of entity names
                   and relative pronouns when the entity is
                  self-identifying and there are values present)
  glevel : 0 .. 24; (controls indentation depth of RT phrases)
  gprojected_out: boolean; (controls comparison phrases)
  attribute_names : StColType;
   operation       : char;
   cols            :  cols_type;
   midnodestack, leftnodestack, rightnodestack : pointerstack;
   s        : string[80]; (general purpose string holder)
   p_table        : P_table_type;
   pro_table      : Pro_table_type;
   row, col       : integer;( row and column index variables for
                               loading tables )
   I              : 0 .. maxint; (loop control variable)
   debug          : BOOLEAN;
   O              : menu_Choice_type; (holds user's menu choice)
   RD             : rd_type;
   dd             : dd_type;
   rdnf, ddnf     : integer; (next free location in dd)
   next_node_id   :node_id_type;

   F              : TEXT ; ( our all-purpose file variable)
   f_name       : File_name_type;
   Main_menu,
   Ops_menu,
   datatype_menu: menu_type;

   R               : relindex_type;
   D               : domindex_type; (dummy variable for initial call of
                                              utter)

{(P)
{PROCEDURES}

FUNCTION getname (VAR i:integer;  s1:string[80]; legals:setofchar):string[80];
VAR     s2      : string[80];
             ch: char;
BEGIN
  WHILE (substr(s1,i,1)=' ' ) DO i:= i+1;
  s2:='';
        ch:=unstr(substr(S1,i,1));
WHILE ch IN legals  DO                                            BEGIN
    s2:= S2 +substr(s1,i,1);  i:=i+1; ch:=unstr(substr(s1,i,1))    END;
getname:= s2;
END;

FUNCTION get_op_token( s: string[80] ): coptype;
VAR i :coptype;
BEGIN
  IF s = '<'  THEN i := Stℓ ELSE
  IF s = '<=' THEN i := Seℓ ELSE
  IF s = '<>' THEN i := Neqℓ ELSE
  IF s = '='  THEN i := Eqℓ ELSE
  IF s = '>=' THEN i := Geℓ ELSE
  IF s = '>'  THEN i := Gtℓ ELSE
                   i := 0;
  get_op_token := i
END; (of get_op_token)
FUNCTION get_att_num( s1:string[80]; r : relindex_type):integer;
VAR
  i:integer;  found: boolean;  s2: string[80]; rdeg:1 .. degmax;
BEGIN
  i:=1; found:=FALSE;
  rdeg:= rd[r].rdegree;
  WHILE (i <= rdeg) AND (NOT(found)) DO   BEGIN
    s2 := rd[r].attribute[i].aname;
    s2:=TRIM(s2);
    s2:=LTRIM(s2);
    found:= (s1=s2);
    IF NOT(found) THEN i:= i + 1          END;
  IF NOT(found) THEN i:= 0;
  get_att_num:=i
END;{of get_att}

FUNCTION get_relation_num( rname:string[80]   ):relindex_type;
VAR i : relindex_type;
    s : string[80];
    found:boolean;
BEGIN
    rname := TRIM(rname);
    rname := LTRIM(rname);
  i:= 1;  found := false;
  WHILE (i < rdnf)  AND ( NOT(found)) DO           BEGIN
     s := rd[i].rname;
        s := TRIM(s);
        s := LTRIM(s);
        found := (rname = s);
        IF NOT found THEN i := i + 1             END;
  IF NOT found THEN i:=0;
  get_relation_num:= i
END;


FUNCTION get_domain_num( dname:string[80]):domindex_type;
VAR i : domindex_type;
    found:boolean;
    s : string[80];
BEGIN
  i:= 1;  found := false;
  dname:=TRIM(dname);
  dname:=LTRIM(dname);
  WHILE (i < ddnf)  AND (NOT(found)) DO           BEGIN
    s:=dd[i].entity_set_name[singular];
    s:=TRIM(s);
    s:=LTRIM(s);
    found := (dname = s);
    IF NOT found THEN i := i + 1             END;
  IF NOT found THEN i:=0;
  get_domain_num:= i
END;


FUNCTION Found_In(X:Col_index_type;
                  Y:Cols_type): BOOLEAN;
VAR I:Col_index_type;
BEGIN
   Found_in:=FALSE;
   FOR I:=1 TO Y.num DO
     IF Y.Cols[I]=X THEN Found_in:=TRUE;
END;

PROCEDURE Load_Menu( VAR  F: TEXT; VAR M:Menu_Type);
BEGIN
WITH M DO
BEGIN
    Choices:=0;
   WHILE NOT EOF(F) DO             BEGIN
     Readln(F, Options[Choices] );
     Choices:= Choices+1;
                                    END;
END;
END; {of Load_Menu}
{***}

FUNCTION Factorial(N:INTEGER):INTEGER;
BEGIN
   IF N=0 THEN Factorial:=1
           ELSE Factorial:=N*Factorial(N-1);
END;
```

```
FUNCTION Copy_node( N:Nodepointer      ): Nodepointer;
VAR
        P       :  Nodepointer;
      i       :  0..24;

 BEGIN
  IF Debug THEN writeln('Copy_node Called.');

  CASE  N^.Typeofnode OF
      Ropnode:
                 New(P, Ropnode);

      Copnode:
                 New(P, Copnode);

      Pnode:
                 New(P, Pnode);

      enode:
                 New(P, Enode);
      Valuenode:
                 New(P, Valuenode);
   END ; {of CASE}
P^:=N^;
WITH N^    DO                                              BEGIN
   FOR i:= 1 TO descendants DO p^.ptr[i] := copy_node (ptr[i]);
                                                               END;{WITH}


 Copy_node := P;
END; {of CopyNode}

FUNCTION Calculate_Order (Permutation, Argument: INTEGER; P: P_Table_Type): INTEGER;
BEGIN
   Calculate_Order := P[Permutation,Argument];
END; {of Calculate_order}
{this must be modified so that it correctly returns the order for
a relation, of any degree, not just of degree 1..4 as it does now}
PROCEDURE Pause;

VAR
   C : Char;

BEGIN
   writeln;
   READLN    ;
   write( Chr(7) );
  writeln;
   WRITE (    'Press RETURN to continue: ' ) ;
   READ (    C ) ;
END;{of Pause}

PROCEDURE Show_Menu ( M: Menu_type);
VAR
   I : 0..MenuMax;
   S : string[80];
BEGIN
writeln;
WITH M DO
BEGIN
   FOR I := 0 TO Choices-1  DO
   BEGIN
    S := Options [I] ;
    writeln(I:3,S);
   END
END;
writeln;
END; {of Show_Menu}

FUNCTION map (pnode : node): Perm_Type;
CONST  maxperms = 24;

VAR  I, k : 0 .. maxint;
     cols  : ARRAY [1..degmax] OF integer;
     row, col : 1..maxint;
     found    : 0..maxint;
{  If a relation's attributes have a one for one relationship
     with the entities of a predication, then this function
    wouldn't be necessary.
    However, when a relation has more attributes than a predication
    has entities, or vice versa, then we must MAP the permutation
    of the relation into(onto?) that of the predication.

    For example, consider the relation R1, with attributes A and b.
    It used to have an attribute C, but this was projected out.
    If we permute R1, how are we to permute its predication?

    R (before projection)  A    B    C
    P                      Ea   Eb   Ec

    It's easy to see that the relation's permutation will be the predication's.

    Now consider       R1  A   B            ( R1 <- R % [A,B] ;)
    with predication P  Ea  Eb  (Ec)

    If we permute R1,    R2 <- R1 %% [B,A]
    how should we 'utter' its predication P ?

    P Eb (Ec) Ea  ?   or
    P Eb  Ea (Ec) ?   or
    P(Ec) Eb  Ea  ?

         Shall we adopt a rule that says, put all projected-out
    entities at the end?  (ie, choose option 2 above)?

    And if there is more than one projected-out entity,
    in what order shall we utter them?

         And consider the opposite problem: the relation with
    several predications, eg

      R     A    B    C
  P1 Ea  Eb
  P2 Eb  Ec
}

BEGIN  WITH  pnode  DO                                 BEGIN
    {We shall chose first pperm such that each active enode
    (those corresponding to existing -- non-projected -- attributes)
    is uttered before any inactive enode.}

   k:= 0;
   IF debug THEN writeln('inside function MAP. descendants = ', descendants:3);
   FOR I:= 1 TO descendants DO
     IF ptr[I]^.attnum > 0 THEN BEGIN
       k := k + 1;
       cols [k] := I;                   END;

                      {the numbers of the active enodes are now in cols
                       from col[1] to col[k]}

    row := 0 ;
    REPEAT
      found := 0;
      row := row + 1;
      FOR col := 1 TO K {k = number of active nodes} DO
        IF p_table [row, col] = cols [col] THEN found := found +1 ;

    UNTIL (found = k)  OR (row > maxperms);

    IF row > maxperms THEN  writeln('error in MAP ');
    map := row; {this permutation has all the active nodes first}
 END                                         END;{of MAP}

 FUNCTION WIDTH(I:INTEGER):INTEGER;
 VAR J, W: INTEGER;
 BEGIN
   J := ABS(I);
   W := 1;

   WHILE W < J DO  W := W*10 ;

   IF J<W THEN W := W-1;
   IF I<0 THEN W := W+1;
 WIDTH := W;
END; {of WIDTH}

PROCEDURE Verbalise(I:Att_index_Type);
BEGIN
  CASE I OF
     1: write(' First ');
     2: Write(' Second ');
     3: Write(' Third ');
     4: Write(' Fourth ');
     5: Write(' Fifth ');
     6: Write(' Sixth ');
  END;
END;{of Verbalise}

FUNCTION Rquery (S:string[80]; Min, Max : REAL): REAL;
VAR R: REAL;
BEGIN
  write(S);
  Readln(R);

  WHILE (R < Min) OR ( R > Max) DO                                    BEGIN
    Writeln('You must  choose a decimal number from ', Min:12:3, ' to ', Max:12:3);
    Write(S);                                                          END;
    Readln(R);
 Rquery := R;

END; {of Rquery}


FUNCTION BQuery (S:String[80]): BOOLEAN;
{ This procedure is designed to put a 'yes or no' type question to
  to user at a terminal, and to get his or her answer, which must
  have a first letter = 'Y', 'y', 'N', 'n'.

  An improvement would be to add a 'skip past leading blanks and puncuation
  marks' feature, (but watch  out for the 'empty line' problem), as at present
  if a user types  one or more spaces before he types Yes, the procedure
  will take it as a 'non-yes' answer.
  An example of its use is:

  VAR
     Help, Finish : Boolean; ( used to get user responses )
     .
     .
     .

  Help := Query ( 'Do you want help?' );
  IF NOT (Help) THEN Finish := Query ( 'Do you want to quit?' );
     .
     .
     .

 }
 VAR
     Response:Char;
 BEGIN
   REPEAT
      writeln(S);
      Write( 'Answer (Y/N): ');
      REPEAT
         READ ( Response); {will read the first character into Response
                            and will skip past any other characters till
                            it gets to the end-of-line}

      UNTIL ( NOT (Response IN [' ']) );
   UNTIL  Response IN ['N','Y','n','y'] ; { don't let 'em go until they answer
                                             yes or no ! ! ! }

   READLN ( INPUT );  { to get past any other rubbish on the line }
   IF ( Response = 'Y' )  OR  ( Response = 'y' )  THEN BQuery := TRUE
   ELSE                                                BQuery := FALSE;
 END ; {of FUNCTION BQuery }


  FUNCTION Squery (S1:string[80]): string[80];
  {takes as input message you want to appear on screen
 appends a colon and carriage return.
 Gets user input, terminated by carriage return.
 If carriage return only is input, returns a single blank.
 Filters out all non-printing characters.
 Allows user to edit his input using BS or CTRL H.}
VAR  S3, S2:string[80];
      i :integer;
BEGIN
    readln; {flush input buffer}
  writeln(S1);
  write(': ');
  Readln(S2);
    S3:='';
    FOR i:=1 TO length(S2) DO BEGIN
      IF (substr(S2,i,1)>=' ' ) AND (substr(S2,i,1) <= '~') THEN
         S3:=S3+substr(S2,i,1)
      ELSE
      IF length(S3)=1 THEN S3:='' ELSE S3:=substr(S3,1,(length(S3)-1))END;
  IF length(s3)=0 THEN squery:=' ' ELSE squery:=s3;
END; {of Squery}

FUNCTION Iquery (S:string[80]; Min, Max : INTEGER): INTEGER;
{returns an integer, within (inclusive) limits Min and Max
 supplied by the programmer. S is an input message.
 Repeats itself if the input integer is outside the limits,
 or if anything other than a digit is typed in.
 A single carriage return is treated as a zero.
 Improvements would be:
   recognise typical errors:  trying to type in a decimal
            number,
                                     'l' for '1', 'O' for '0'
   allow an escape (like '-') to bypass max/min checking
   and return a '1'                                             }
VAR i, len, sign, n, digit, Minwidth, Maxwidth : INTEGER;
         ch : char;
     s1 : string[80];
BEGIN
  minwidth:=3; maxwidth:=6;{temp. fix -- change later}
  REPEAT
     s1:='';
      s1:=squery(S);
      s1:=trim(ltrim(s1));
      IF s1='' THEN s1:='0';
      sign:=1;
      IF substr(s1,1,1)='-' THEN BEGIN
         sign:=-1;
         s1:=substr(s1,2, length(s1)-1) END
      ELSE
      IF substr(s1,1,1)='+' THEN
         s1:=substr(s1,2,length(s1)-1);
      n:=0;
      len:=length(s1);
      i:=1;
      WHILE i <= len DO                                  BEGIN
         ch := unstr(substr(s1,i,1));
         IF  NOT(ch IN ['0'..'9']) THEN BEGIN
           writeln('OOPS! ''',ch,'''',' is not a digit.');
           writeln('Type in a number, using 0 1 2 ... 8 9');          END
           i := len + 1; n:=min-1
           ELSE                                    BEGIN
            digit := ord(ch)-ord('0');
               n:= 10*n + digit               END; i:= i+1   END;
        n:= n*sign;

        IF  (n < min)  OR  (n>max) THEN
         writeln('You must  choose a whole number from ', Min:Minwidth, ' to ', Max:Maxwidth);
     UNTIL (n>=min) AND (n<=max);
     Iquery := n;

 END;{of Iquery}
 PROCEDURE push( p:nodepointer; VAR S:pointerstack);
 BEGIN WITH S DO BEGIN
   IF nf=stackmax + 1 THEN write('error in proc. push -- stack full')
   ELSE pointers[nf]:=p;
   nf := nf + 1;
 END END;

 FUNCTION pop( VAR S:pointerstack):nodepointer;
 BEGIN WITH S DO BEGIN
   nf:=nf-1;
  IF nf<1 THEN write ('error in pop, trying to pop an empty stack')
  ELSE pop:= pointers[nf]
END END;

PROCEDURE findnodes ( ptr : nodepointer;
                      typesearchedfor : nodetype;
                      VAR S : pointerstack );
VAR
    I : 1 .. degmax;
BEGIN
    IF ptr^.typeofnode =typesearchedfor THEN  push(ptr, S);
    WITH ptr^ DO BEGIN
    IF descendants > 0      {if it's not a leaf}
    THEN
       FOR I:= 1 TO descendants DO   findnodes( ptr[I], typesearchedfor,S);
END
END;

PROCEDURE canonise( pointer : nodepointer ); {a pointer to U  or ^  node}
VAR
  nodeholder : nodepointer;
BEGIN   WITH  pointer^ DO                                                BEGIN
  IF ptr[1]^.ptr[1]^.value >
     ptr[2]^.ptr[1]^.value                                             BEGIN
  THEN
    IF debug THEN write('swapping leaves');
    nodeholder := ptr[2];
    ptr[2] := ptr[1];                                                  END
    ptr[1] := nodeholder                                                      END;
END

PROCEDURE Lower( VAR pointer:nodepointer ); {pointer to Pnode}
VAR
    temp1, temp2 : nodepointer;
BEGIN  WITH pointer^ {pnode} DO                                           BEGIN
   temp1 := copy_node( ptr[1]); {ptr to enode which will be raised}
   ptr[1]^.dummy:= YES;
   {prole := subordinate; } {so it is uttered like a comparison node}
   WITH temp1^  {enode to be raised}  DO                               BEGIN
     IF descendants = 0 {no QGraph} THEN                              BEGIN
        descendants := 1; {but we will attach one}
        ptr[1] := pointer;{the pnode,  } END
     ELSE {descendants=1, ie it has QGraph already}                  BEGIN
        temp2:=ptr[1]; {store its QGraph}
        NEW(ptr[1], Ropnode); {create an AND node}
        WITH ptr[1]^ DO                                            BEGIN
          typeofnode := Ropnode;
          op         := andx;
          ptr[1] := temp2; {old QGraph}
          ptr[2] := pointer; {new, lowered, pnode-enode graph}   END END  END;
  pointer:=temp1; {now pointer points to raised enode}                        END;
END


FUNCTION get_rel_pro(whichcase:case_type; D:domindex_type): string[80];
VAR rep_name_holder : string [80];
BEGIN

        IF (dd[D].RepData.Represented=Yes) THEN     BEGIN
           rep_name_holder :=  dd[D].RepData.Repname[singular];
           get_rel_pro := 'whose ' + rep_name_holder      END
        ELSE
           get_rel_pro := Pro_table[ dd[D].animate_value, whichcase];
       IF debug THEN writeln('anim val, case =', dd[D].animate_value, whichcase);


 END; {of get_rel_pro}

 FUNCTION compare_trees(ptr1, ptr2 : nodepointer; nodeset : setofnodetypes):boolean;
 { For predication-identity test, let nodeset := [ropnode, pnode, enode]
   For total-identity test,       let nodeset := [nodetype]}
 VAR   r : boolean;
       i : integer;
 BEGIN
    r:= TRUE;
    IF ptr1^.typeofnode = ptr2^.typeofnode THEN                       BEGIN
      CASE ptr1^.typeofnode OF
          ropnode:r:= ptr1^.op = ptr2^.op;
          copnode: r := ptr1^.cop = ptr2^.cop;{should never get to a comp. node}
        pnode: r:= (ptr1^.pred_id = ptr2^.pred_id)
                                AND
                        (ptr1^.pperm  = ptr2^.pperm )
                                AND
                        (ptr1^.psense = ptr2^.psense);
       enode:r:= ptr1^.entdomain_id = ptr2^.entdomain_id ;
       valuenode: r := ptr1^.value = ptr2^.value ;
    END {of CASE};

   IF r  {these nodes are the same, so check descendants} THEN        BEGIN
    i:=1;
    IF (ptr1^.typeofnode IN nodeset ) THEN
    WHILE ( r AND (i <= ptr1^.descendants) ) DO       BEGIN
       r:= compare_trees(ptr1^.ptr[i], ptr2^.ptr[i], nodeset);
       i := i + 1                                              END      END
   END {of IF nodetypes same }
 ELSE  r:= FALSE;
 compare_trees := r
END;


PROCEDURE Show_Domain(D:integer);
VAR
   S1, S2: string[80];
   I      : INTEGER;                                                 BEGIN
BEGIN   WITH  dd[D] DO
   S1:= Entity_set_name[singular];
   S2 := Entity_Set_name[plural];
   writeln;
   writeln('DOMAIN: ', D:3,' ', S1,' ', S2);

   IF Animate_value = Animate
   THEN writeln('      Animate')
   ELSE writeln('      Inanimate');

   IF Self_ID = Yes
   THEN writeln('      Self-identifying')
   ELSE writeln('      Not Self-identifying');
                                                        BEGIN
   WITH RepData DO
   IF Represented = Yes THEN                         BEGIN
      s1:= RepName[singular];  s2:= RepName[plural];
        writeln('      Represented by: ', S1,'/', S2)   END
   ELSE writeln('      Self-represented')                END;
                                                                  BEGIN
   WITH RangeData DO
   write('      Data type: ');
   CASE Datatype OF
    Bool: write(' Boolean');
    Int : write(' Integer (from ',IMin:3,' to ',IMax:3,')');
    Rel : write(' Real    (from ',RMin:8:8, ' to ',RMax:8:8, ')');
    Cha : write(' Single characters');
  Str : write('Strings.');
    Dat : write(' Dates ');

  END; {of CASE}                                  { RangeData : } END;
                                                    BEGIN
       WITH OrderData DO
         IF Ordered = Yes THEN                 BEGIN
            writeln('Comparison phrases: ');
           FOR I:= STE TO GTE  DC         BEGIN
                S1 := CompPhrases[I];
                writeln(S1);                END END      END;
                                                           BEGIN
       WITH ValueSetData DO
         IF Limited = Yes THEN                  BEGIN
         WITH Values DO                  BEGIN
            FOR I:=1 TO Count DO  BEGIN
                S1:= String[I];
                writeln(S1);        END    END     END     END;
                                                  END; {of Show_domain}
 END{of WITH}

 PROCEDURE show_relation(R:integer);
 VAR    S: string;
        I, J: 0..24;
        key : boolean;                                                   BEGIN
 BEGIN          WITH  RD[R] DO
     IF debug THEN writeln('RDegree = ', RDegree);
    S:=Rname;
  writeln( 'Relation name: ', S:20);

  {list each attribute, indicate ('*') if it is a KEY attribute}

  write('Attributes: ');

  FOR I:=1 TO RDegree DO                         BEGIN
    S:=Attribute[I].Aname;
    Write(S: 10);
    key := FALSE ;
    FOR J:= 1 TO Key_Cols.Num DO
      IF I=Key_Cols.Cols[J] THEN key := TRUE;
    IF key THEN write ('* ') ELSE write('  ')  END;

  writeln;

  {now show the DOMAIN each attribute is drawn from}

  write('Domains:     ');

  FOR I:= 1 TO RDegree DO                                  BEGIN
    S:= dd[  Attribute[I].Attdomain_id ].Entity_set_name[Singular];
    Write(S:12)                                             END;

END{of WITH,                       of show_a_relation }          END;
PROCEDURE Get_File_Name ( VAR File_name: File_name_Type);
VAR I:INTEGER;
  Ch: CHAR;
BEGIN
   Write('File name: ');

   I:=0;
   READ(Ch);
   WHILE NOT EOLN(INPUT) DO
      BEGIN
        I:=I+1;
        File_name[I]:= Ch;
        READ(Ch);
      END;
      I:=I+1;
      File_name[I]:= Ch;
   FOR I := I + 1 TO 32 DO File_name[I] := ' '; { Input trailing blanks }
END;


FUNCTION Load_node( VAR F: TEXT      ): Nodepointer;
VAR
         P : Nodepointer;
       I,J : 0..24;
     Sense : Sense_type;
    Number : Number_type;
  Permutations : 1..24;
  This_perm, This_phrase : 1..24;
         S : String[80];
         N : Nodetype;
BEGIN
  IF Debug THEN writeln('Load_node Called.');
  Readln(F, N); {find out the type of node we'll be loading}
  IF Debug THEN writeln('Nodetype read okay, now to create a node.');

  CASE  N OF
     Ropnode: BEGIN
           New(P, Ropnode);
           END;


     Copnode: BEGIN
           New(P, Copnode);
           END;


     Pnode: BEGIN
           New(P, Pnode);
           END;


     Enode: BEGIN
           New(P, Enode);
           END;
   Valuenode:  New(P, Valuenode);

END ; {of CASE}
      {now we have created a node of the proper type let's fill in the values}
IF Debug THEN Writeln('New node created okay, now to fill it in.');
```

```
WITH P^ DO                                   BEGIN

Readln(F, Descendants); {number of times this procedure will call itself from this level}
IF Debug THEN Writeln('This node has ',Descendants:3,' descendant nodes.');

 Typeofnode:=N;
 CASE    Typeofnode OF

     Kopnode:                        BEGIN
       Readln(F, Op);               END;

     Copnode:                                BEGIN
        Readln(F, Cop);
        readln(F, cDegree);          END;

     Pnode: BEGIN
```

```
        Readln(F, Pred_id);
        Readln(F, Pperm);
        Readln(F, Pred_type);
        Readln(F, Psense);
        Readln(F, Pnumber);

        IF Debug THEN Writeln('Psense and Pnumber read okay, =',Psense:3,Pnumber:3);
        Permutations:=Factorial(Descendants);
        FOR This_perm:=1 TO Permutations DO
          FOR This_phrase := 1 TO Descendants-1 DO
          WITH Alt_sentences[This_perm, This_phrase] DO              BEGIN
            Readln(F,Main);
            CASE Main OF
                    No: BEGIN Readln(F,S); Case_phrase:=S END;
                    Yes:  BEGIN
                            FOR Sense:=Positive DOWNTO Negative DO
                              FOR Number:=Singular TO Plural DO BEGIN
                                 Readln(F,S);
                                 Alt_phrases[Sense, Number] := S   END;
                          END; {of Yes}
             END; {of CASE}
                                                            END; {of double loop and WITH}
     IF Debug THEN Writeln('Now outside of phrases loop, preparing to load descendants.');

                                  {Pnode: }                 END;

     Enode: BEGIN

         readln(F, attnum);
         IF debug THEN writeln('attnum = ', attnum);
         readln(F,Entdomain_id);
         IF debug THEN    write('Entdomain id = ', Entdomain_id);
         readln(F, Dummy);
         IF debug THEN writeln('dummy = ', Dummy);
         readln(F, edegree);
         IF debug THEN writeln('edegree = ', edegree);
         readln(F, node_id);
         IF debug THEN writeln('node_id = ', node_id:3);

         readln(F, role[singular]);
         readln(F, role[plural]);
         IF debug THEN writeln('roles: ', role[singular], role[plural]);
         readln(F, trimmed);
         IF debug THEN writeln('trimmed = ', trimmed:3);
         IF trimmed = yes THEN readln(F, shortdesc);   END;

       Valuenode: BEGIN
        Readln(F, Value);

             END;

    END ; {of CASE}

   IF debug THEN writeln('now loading descendants of this node.');
    FOR i:= 1 TO descendants DO   ptr[i] := load_node( F );
                                                        END; {of WITH}


    Load_node := P;
  END;
PROCEDURE load_dict( VAR f: text);
VAR pcount, dnum, rnum,  I,  J, K, L:INTEGER;
   s : STRING[80]; (debug)
        Sense : Sense_type;
        Number: Number_type;
BEGIN
   readln(F, next_node_id); {must be 0 for initalisation}
   readln(F, dnum); {get the number of domains stored here; must be 0 for
                   initialising system}
   ddnf := dnum + 1;
writeln('ddnf now = ', ddnf);
   FOR i := 1 TO dnum DO WITH dd[i] DO                                BEGIN

      readln(F, Entity_set_name[singular]);
      readln(F, Entity_set_name[ Plural]);

      readln(F, Animate_Value);
readln(F, Self_ID);
WITH RepData DO                                         BEGIN
   readln(F, Represented);
   IF Represented = Yes     THEN     BEGIN
      readln(F, RepName[Singular]);
      readln(F, RepName[Plural]);
                                            END;   END; {of RepData}

WITH RangeData DO                                             BEGIN
readln(F, Datatype);
   CASE Datatype OF
      Bool, Cha, Str, Dat :BEGIN
                           END;
      Rel         : BEGIN
                         readln(F, RMin);
                         readln(F, RMax);
                      END;
      Int   : BEGIN
                   readln(F, IMin);
                   readln(F, IMax);
                END;
   END; {of CASE}                                     END; {of RangeData}

WITH OrderData  DO                                            BEGIN
   readln(F, Ordered);
   IF Ordered = No THEN pcount:=2 ELSE Pcount := 6;
   FOR J:= 1 TO pcount DO                                      {of OrderData}
        readln(F, CompPhrases[J]);                           END; {of OrderData}

WITH ValueSetData     DO                                     BEGIN
   readln(F, Limited);
   IF Limited = Yes THEN               BEGIN
      readln(F, Values.Count);
      FOR J := 1 TO Values.Count DO
        readln(F, Values.String[J]);    END;     END; {of ValueSetData}
                                                    { Loop and WITH : } END;



  { load the relation definitions }


readln(F, rnum);  { same method as for loading domains }
FOR I := 1 TO rnum DO                                                BEGIN
WITH RD [I] DO

    readln (F, RName);
    readln (F, RDegree);
    readln (F, Key_cols.num);
    FOR J:=1 TO Key_cols.num DO  readln(F, Key_cols.cols[J]);
    readln (F, RPerm);
    readln (F, Preds);
    FOR J:= 1 To RDegree DO
       WITH Attribute [J] DO    BEGIN
         readln(F, Aname);
         readln(F, Attdomain_id)   END;
    {Now load the predications}

    FOR J:= 1 TO Preds DO BEGIN
       Ptr[J] := Load_Node(F);END;

                                                      {Loop and WITH}END;

   writeln('rdnf = ', rdnf);
   rdnf:= rnum  + 1;
   writeln('rdnf updated by rnum, now = ', rdnf);

   Close(F);
END; {of Load_Dict}


PROCEDURE Dump_Node ( VAR F:Text;{where we are dumping them}
                          N:Nodepointer   );

VAR I,J: 1..24;
    This_perm, This_phrase : 1..24; { 1 to MAX of DegMax|PermMax }
```

```
     Permutations : 1..24;
    S: String[80];
    sense : negative..positive;
    number : singular .. plural;(can we eliminate these two?)
BEGIN
  IF Debug THEN Writeln('Dump_Node called.');
  WITH N^ DO                           BEGIN
   Write(F, Typeofnode);
   IF Debug THEN Write(F, ' type of node. 1 = Rop, 2 = Cop, 3 = Pred, 4 = Entity, 5 = Value');
   writeln(F);
   Write(F, Descendants); IF Debug THEN Write(F, '   the number of descendant nodes.');
   Writeln(F);
   IF Debug THEN Writeln('Dumping a node of type ', Typeofnode);
   CASE  Typeofnode    OF

       Ropnode:             BEGIN
          write(F, Op);
          IF debug THEN write(F,' an Ropnode');
          writeln(F)         END;(of _Node)

       Copnode:        BEGIN
         write( F, Cop);
         IF debug THEN write(F,' a Copnode.');
         writeln(F);
         write(F, CDegree);
         IF debug THEN write(F,' its Edegree');
         writeln(F);    END; (of Copnode)

       Pnode:                                                 BEGIN
             write(F,Pred_id);
             IF debug THEN write(F,'    internal identifier of pred.');
             writeln(F);
             write(F, PPerm); IF Debug THEN write(F, ' the permutation of this predication.');
             Writeln(F);
             Write(F,Pred_type); IF Debug THEN Write(F,' an attribute predication or a relationship');
             Writeln(F);
(*)

             Permutations:=Factorial(Descendants);
             IF Debug THEN Writeln('Now getting ready to dump predication phrases');

            write(F, PSense); IF Debug THEN write(F, ' sense: 0 = negative, 1 = positive');
            Writeln(F);

            write(F, Pnumber); IF Debug THEN write(F, ' number: 0 = singular, 1 = plural');
            Writeln(F);


            IF Debug THEN Writeln('Now dumping predication phrases');
           FOR This_perm:=1 TO Permutations DO
             FOR This_phrase := 1 TO Descendants-1 DO
             WITH Alt_sentences[This_perm, This_phrase] DO             BEGIN
               Write(F,Main);
               IF Debug THEN write(F, '    0 =  a case phrase,  1 = Main Pred. Phrase.');
               Writeln(F);
               CASE Main OF
                  No:  BEGIN S:=Case_phrase; Writeln(F,S) END;
                  Yes:  BEGIN
                         FOR Sense:=Positive DOWNTO Negative DO
                           FOR Number:=Singular TO Plural DO BEGIN
                               S:=Alt_phrases[Sense, Number];
                               Writeln(F,S);                   END;
                       END; (of Yes)
                 END; (of CASE)
                                                       END; (of double loop and WITH)
                                                               END;(of Pnode)


      Enode:                          BEGIN
         Write(F, Attnum); IF Debug THEN write(F, ' the attribute corresponding to this entity');
         Writeln(F);
         write(F, Entdomain_id);
         IF Debug THEN write (F,'    domain number of this entity');
         writeln(F);
         write(F, dummy);  IF debug THEN write(F, '    1 = dummy node');
         writeln(F);
         write(F, cdegree); IF Debug THEN Write(F, ' the ''degree'' (one, many, all) of this entity');
         Writeln(F);
         write(F, node_id); IF debug THEN write(F, '    node_id number ');
         writeln(F);
```

```
          writeln(F, Role[singular]);
          writeln(F, Role[plural]);
          write(F, trimmed); IF debug THEN write(F,' 1 = trimmed, 0 not');
          writeln(F);
          IF trimmed=yes THEN writeln(F, shortdesc);    END;

      Valuenode:                      BEGIN
           {a Value node has no descendants}
              writeln( F, Value);  END;


    END; {of CASE}


    IF Debug THEN Writeln('Now dumping the descendants.');
    FOR I:= 1 TO Descendants DO
       Dump_Node(F, Ptr[I] );


END {of WITH}
END; {of Dump_Node}


PROCEDURE dump_dict(VAR F: text);
VAR pcount, Dnum, Rnum,  I,  J, K  :0..24;
       S:STRING;
BEGIN
 IF Debug THEN Writeln('Dumping Data to NEW.dd.DAT');
 IF Debug THEN Writeln('ddnf-1 = ', ddnf-1);
 write(F, next_node_id);
 IF debug THEN write(F, '  the id number of next enode');
 writeln(F);
 write(F, ddnf-1); {put the number of domains stored here; must be 0 for
                     initialising system}
 IF Debug THEN write(F,'  the number of domains/entities stored.');
 writeln(F);
 FOR I := 1 TO ddnf-1 DO
 WITH dd[ I ]  DO                                              BEGIN
   {}IF Debug THEN Writeln('Trying to dump number ', I, ' domain');
   Writeln(F, Entity_set_name[ Singular]);
   Writeln(F, Entity_set_name [ Plural]);
   write(F, Animate_Value);IF Debug THEN write(F, '  animate = 1, 0 otherwise');
   Writeln(F);
   write(F, Self_id); IF Debug THEN write(F,' 1= self-identifying, 0 not');
   Writeln(F);
   IF Debug THEN writeln('New trying to dump RepData.');
   WITH RepData   DO                                 BEGIN
    write(F, Represented);
    IF Debug THEN Write(F,'  1 = represented by another entity, 0 = not.');
    Writeln(F);
    IF Represented = Yes    THEN     BEGIN
      Writeln(F, RepName[Singular]);
      Writeln(F, RepName[Plural]);  END {of Repdata} END;

   IF Debug THEN Writeln('Now trying to dump RangeData.');
   WITH RangeData DO                              BEGIN
    Write(F, Datatype);
    IF Debug THEN Write(F,'  Datatype of reps (or self). 0 = boolean, 1 = integer,');
    writeln(F);
    CASE Datatype OF
      Bool, Cha, Str :BEGIN
               END;
      Rel   : BEGIN
                write (F, RMin);IF Debug THEN write(F,' Minimum value');
                writeln(F);
                write (F, RMax);IF Debug THEN write(F,' Maximum value');
                writeln(F);
              END;
      Int  : BEGIN
               write(F, IMin);IF Debug THEN write(F, '  Minimum value');
               writeln(F);
               write(F, IMax); IF Debug THEN write(F, '  Maximum value.');
               writeln(F);
             END;
      Dat: BEGIN  END;
   END; { of CASE }                            END; {of RangeData}

   IF Debug THEN Writeln('Now trying to dump Orderdata.');
   WITH OrderData DO                  BEGIN
    Write(F, Ordered);IF Debug THEN write(F, '  1 = ordered, 0 = not.');
```

```
  Writeln(F);
   IF Ordered  = No THEN pcount := 2 ELSE pcount := 6;
      FOR J:= 1 TO pcount DO     BEGIN
        S := CompPhrases [J] ;
        Writeln(F, S);             END    END; {of OrderData}

  WITH ValueSetData  DO                     BEGIN
    write(F, Limited); IF Debug THEN Write(F,'  1 = Limited value set, 0 = not.');
    Writeln(F);
    IF Limited = Yes THEN            BEGIN
      Write(F, Values.Count);
      IF debug THEN write(F,' values count');
      writeln(F);
      FOR J := 1 TO Values.Count DO     BEGIN
           S := Values.String[J];
           Writeln(F, S);                END END; END; {of ValueSetData}

  {} IF Debug THEN Writeln('number', I, ' domain dumped');

                                                       END;{of Loop}

  { dump the relation definitions }

IF Debug THEN Writeln('Now trying to dump Relations data.');
write(F, ranf-1);  { same method as for dumping domains }
IF Debug THEN write(F, '  the number of relations stored in the dictionary.');
writeln(F);

FOR I := 1 TO ranf-1 DO                             WITH RD[I] DO BEGIN
    IF debug THEN writeln('Now dumping relation ',I);

    writeln (F, RName);

    Write (F, RDegree);
    IF Debug THEN      write(F,'  the degree of this relation.');
    Writeln(F);

    Write (F, Key_cols.num);
    IF Debug THEN      write(F, '  the number of atts making up the key, and which they are:');
    Writeln(F);

   FOR J:=1 TO Key_cols.num DO  Writeln(F, Key_cols.cols[J]);
   write (F, RPerm);
    IF Debug THEN      Write(F, '  the particular permutation of this relation.');
    Writeln(F);

    Write (F, Preds);
    IF Debug THEN     Write(F,'  the number of predications embodied in this relation.');
    Writeln(F);

    IF Debug THEN Writeln('Header info dumped, now beginning Attribute data.');

    FOR J:= 1 To RDegree DO
       WITH Attribute [J] DO              BEGIN
          Writeln(F, Aname);
          write(F, Attdomain_id);
          IF debug THEN write(F,' domain number of above attribute.');
          Writeln(F)                 END;

    IF Debug THEN Writeln('Now trying to dump predications.');
    FOR J:=1 TO Preds DO
       Dump_Node(F, Ptr[J]);                          {I-Loop and WITH : } END;
END;{of Dump_Dict}

PROCEDURE create_domain;
VAR
  S1:STRING;

BEGIN
WITH dd[ddnf] DO
BEGIN
   Write('Domain name (singular): '); READLN(S1);
   Entity_set_name [ singular ] := S1;
   Write('Domain name    (plural): '); READLN (S1);
   Entity_set_name [ plural ] := S1 ;

   IF BQuery('Are the members of this domain animate?') THEN Animate_value :=Animate
   ELSE Animate_value := Inanimate;

   IF BQuery('Are the members of this domain self-identifying?')
   THEN  Self_ID:=Yes
   ELSE  Self_ID:=No;

   WITH RepData DO                                                      BEGIN
     IF Bquery(
'Will this domain be represented in the database by data values of another domain?') THEN  BEGIN
           Represented := Yes;
           RepName[ singular] :=Squery('Singular representative name: ');
           RepName[Plural] :=Squery('Plural   representative name: ');  END
      ELSE Represented := No;                          END; {of RepData}

    WITH RangeData DO                                          BEGIN
    REPEAT
       Show_Menu( Datatype_Menu);
       Datatype := Iquery('Data type code: ', Bool, Dat);
    UNTIL ( Datatype>=Bool) AND (Datatype<=Dat);

    IF Debug THEN Writeln('Datatype read was ', Datatype);

    CASE Datatype OF
      Bool : BEGIN    END;
      Int   : BEGIN
        IMin := Iquery('Minimum permitted value: ', -Maxint, Maxint);
        IMax := Iquery('Maximum permitted value: ', IMin, Maxint)   END;
      Rel         : BEGIN
        RMin := RQuery('Minimum permitted value: ',-Maxreal, Maxreal);
        RMax := RQuery('Maximum permitted value: ',  RMin,    Maxreal) END;
      Cha, Str,  Dat: BEGIN   END;

     END; {of CASE}                             END; {of RangeData}
                                                          BEGIN
 WITH ValueSetData DO
  IF BQuery('Will this domain (or its representatives) range over a limited value set?') THEN
  BEGIN
     Limited := Yes;
      {Solicit the value set and store it}
     Writeln('not yet implemented');
  END
  ELSE  Limited := No;                          END; {of ValueSetData}

   WITH OrderData DO                             BEGIN
   IF
   NOT (BQuery('Are members of this domain strictly ordered?') )
   THEN                              BEGIN
          Ordered := No;
          Writeln('Input the appropriate phrases for each');
          Writeln('of these comparison operations: ');
          { write('<  '); Readln(S1);
          CompPhrases[Stℓ]:=S1;
          write('<= '); Readln(S1);
          CompPhrases[Seℓ]:=S1;}
          Write('<> '); Readln(S1);
          CompPhrases[Neqℓ]:=S1;
          Write('=  '); Readln(S1);
          CompPhrases[Eqℓ]:=S1;
          {Write('>= '); Readln(S1);
          CompPhrases[Geℓ]:=S1;
          Write('>  '); Readln(S1);
          CompPhrases[Gtℓ]:=S1;}     END
  ELSE                                                        BEGIN
      CompPhrases[Stℓ] := 'less than ';
      CompPhrases[Neqℓ]:='does not equal ';
      CompPhrases[Eqℓ]:='equals ';
      CompPhrases[Seℓ]:='less than, or equal to ';
      CompPhrases[Geℓ]:='greater than, or equal to ';
      CompPhrases[Gtℓ]:='greater than ';
                                                                  END END; {of Orderdata}

    ddnf:=ddnf+1;
    Writeln('Domain now added to dictionary');
  END {of WITH} END; {of Create_Domain}


  PROCEDURE create_relation;
  VAR     S1,S,N: STRING[80];
```

```
        Permutations,  Order, I,J,K,  D:0..120;
        CPrm, Cphr: 1..24;{index variables for looping
                                through all variations of predication}
        MPP : 1..DegMax; {Main Predication Phrase}
        Sense : Sense_Type;
        Number : Number_type;

BEGIN  IF Debug THEN Writeln('crerel, rdnf =', rdnf:3);   WITH  RD[runf] DO  BEGIN
  S1 := Squery('Relation name.');
  RD[rdnf].Rname:=S1;
  writeln('I read ', RD[rdnf].Rname:20);
  {WHILE Locate_Relation (Rname, Rdic, rdnf) <> 0 DO BEGIN
    Writeln(''Rname'', ' is the name of a relation which already exists.');
    writelen('Use a different name.');
    S1 := Squery('Relation name: ');            END;
  }

  RDegree := Iquery('Degree: ', 1, DegMax);

  Key_cols.num:=Iquery('How many attributes in the key: ',1, RDegree);
  IF Key_cols.num < RDegree THEN BEGIN
    writeln('Input the number of each key attribute.');
    FOR I:=1 TO Key_cols.num DO
       Key_cols.cols[I]:=IQuery('Next attribute (number): ',1,Rdegree); {add check for duplicates} END
    ELSE
       FOR I:= 1 TO RDegree DO Key_cols.cols[I]:=I;

  RPerm := 1 ;
           {Get each attribute}

   FOR I:=1 TO RDegree DO  WITH Attribute[I] DO           BEGIN

   Verbalise(I);
   S1 := Squery(' attribute name: ');
   Aname:=S1;
   {WHILE  Locate_Att(Aname, Rdic, rdnf, I-1) <> 0 DO BEGIN
      Writeln('That name has already been used in this relation. Try again.');
      S1 := Squery('           Attribute name: ');  END; }

   IF debug THEN writeln('ddNF = ', ddNF:3);

   Attdomain_id := Iquery('domain number for this attribute: ',1,ddNF-1);
  IF debug THEN writeln('domain ', attdomain_id:3,' is ',dd[attdomain_id].entity_set_name[singular]);
{        IF debug THEN writeln('typeofnode, node_id =', typeofnode:3, node_id:3);}
{        IF debug THEN writeln('Entdomain_id = ',Entdomain_id:3);}
   {WHILE Locate_domain(S1, dd, ddNF) = 0 DO        BEGIN
      writeln('A domain of this name does not exist.');
      Writeln('Existing domains are: ');
      FOR J:=1 TO ddNF-1 DO
        Writeln(<domain name>[J]);

      domain := Locate_domain(S1, dd, ddNF);                END;}
                            {I-loop and WITH  }                    END;

   { now create graph }




  IF Debug THEN writeln('Preparing to create graph.');
```

```
Preds := 1 ; {every base relation will have only one predication --
             this is equivalent to restraining all base relations
             to be held in reduced form}
FOR I := 1 TO MaxPreds Do ptr [ I ] := NIL ;

New(Ptr[1], Pnode); {Create a Predication node, pointed to by 'root' of relation
                     as this is a base relation's predication,
                     its Pdegree will = Rdegree of the relation}

 IF Debug THEN writeln('New Pnode created okay.');
{       IF debug THEN writeln('typeofnode, node_id =', typeofnode:3, node_id:3);}
{       IF debug THEN writeln('Entdomain_id = ',Entdomain_id:3);}

ptr[1]^.descendants :=RDegree;
ptr[1]^.pperm:= 1      ; {a base relation does not need to be mapped}
WITH Ptr[1]^  DO {with the Pnode we've just created...}           BEGIN
  IF debug THEN writeln('mapping completed, pperm = ', pperm);
  Typeofnode:=Pnode;
  Pred_id := rdnf; {takes location of base relation}
  Psense := Positive;  {default}
  Pnumber:= Singular;  {default}

      IF debug THEN writeln('typeofnode, node_id =', typeofnode:3, node_id:3);
      IF debug THEN writeln('Entdomain_id = ',Entdomain_id:3);
  IF Debug THEN Writeln('Descendants =, Rperm = , PPerm = , ', Descendants, Rperm,PPerm);

  FOR I := 1 TO Descendants    DO   {create each entity node}  BEGIN
    New( Ptr[I], Enode);
    WITH Ptr[I]^ DO            BEGIN
     IF debug THEN writeln('creating  Entity node number ', I:2);
     Typeofnode := Enode;
     Verbalise(I); writeln(' entity: ');

     node_id := next_node_id;
     next_node_id := next_node_id +1;
     IF debug THEN writeln('node_id = ', node_id:3);

     IF Bquery('Will this entity play a distinct role in this relation?')
     THEN                                BEGIN
       S := Squery('Singular Role Name?');
       Role[singular] := S;
       S := Squery('  Plural Role Name?');
       Role[plural]   := S                 END
     ELSE BEGIN Role[singular]:='none'; Role[plural]:='none' END;

     Attnum := I ; {the attribute this entity refers to }
    Entdomain_id := Attribute[I].Attdomain_id;
     IF debug THEN writeln('Entdomain_id = ',Entdomain_id:3, dd[Entdomain_id].entity_set_name[singular]);
     dummy := no;
     IF Found_in (I, Key_cols) THEN  EDegree := many
                               ELSE  EDegree := one ;
     IF debug THEN writeln('edegree = ', edegree:3);

     Descendants := 0; {for a base relation}
     FOR J:=1 TO DegMax DO   Ptr[J]:=NIL;
                                                      END {I-loop and WITH}  END;


          {Now get the 'utterances' for each permutation}

   {Here we want to generate all the permutations  of a relationship
    So, for a degree three relationship, say, X 'supplies' Y 'to' Z
    We want to solicit from the user all 6 permutations of this phrase.
    So we want to  get      ___ X _____ Y _____ Z ___
                            ___ X _____ Z _____ Y ___
                            ___ Y _____ X _____ Z ___
                            ___ Y _____ Z _____ X ___
                            ___ Z _____ X _____ Y ___
                            ___ Z _____ Y _____ X ___

    We will do this by enunciating the X, Y, Z domain names  and inviting the
    user to 'fill in the blanks'}


   IF RDegree = 2 THEN                                 BEGIN
      MPP:=1;
      IF Bquery('Is this an Entity/Attribute relationship?')
        THEN Pred_type:=Ent_Att
        ELSE Pred_type:=Ent_Ent                                    END
  ELSE  { RDegree > 2 , SD}
      MPP := Iquery('Number of main predication phrase?',1,RDegree-1);

  Permutations:= Factorial(RDegree);
  FOR CPrm := 1 TO Permutations DO                                                  BEGIN
    Writeln;
    FOR Cphr := 1 TO RDegree   DO                                      BEGIN
      Order := Calculate_Order (CPrm, Cphr, P_Table);
       IF debug THEN  writeln('CPrm, Cphr, Order ,Entdomain_id = ', Cprm:3,Cphr:3,Order:3,Entdomain_id:3);
       IF debug THEN writeln('typeofnode, node_id =', typeofnode:3, node_id:3);
       IF debug THEN writeln('Entdomain_id = ',Entdomain_id:3);
     (*)
       IF debug THEN writeln('Ptr[Order]^.Entdomain_id = ', Ptr[Order]^.Entdomain_id);
      IF debug THEN writeln('Ptr[Order]^.node_id = ', ptr[order]^.node_id:3);
       IF debug THEN writeln('ptr[order]^.typeofnode = ', ptr[order]^.Typeofnode);
       S:= dd[Ptr[Order]^.Entdomain_id].Entity_set_name[singular];
       write(S);
       S:= Ptr[Order]^.Role[singular];
       IF s<>'none' THEN write('(acting as a ', S, ')');
       IF Cphr<RDegree THEN
         write(' < phrase ', Cphr:1, ' > ');
                                                      ( Cphr-loop) END;


   writeln;
   FOR Cphr := 1 TO Descendants-1 DO                                        BEGIN
     IF Cphr<>MPP THEN   (we are dealing with a case phrase)                 BEGIN
       Alt_sentences[CPrm,Cphr].Main:=No;
       Write('Phrase ',Cphr:2);
       S:=Squery('(a Case phrase).');
       Alt_sentences[CPrm,Cphr].Case_phrase:=S;                   END
     ELSE  (it's the main pred. phrase, so get all 4 variants)        BEGIN
       Alt_sentences[CPrm,Cphr].Main:=Yes;
       FOR Sense:=Positive DOWNTO Negative DO                                  BEGIN
         IF Sense=Negative THEN S:='Negative '
                                ELSE S:='Positive ';
         FOR Number:=Singular TO Plural DO    BEGIN
           IF Number = Singular
           THEN N :='   Singular '
           ELSE N :='   Plural ';
           S1:=Squery(S+N+'Phrase.');
           Alt_sentences[CPrm,Cphr].Alt_phrases[Sense,Number]:=S1 END   END END    END;
                                              (CPrm-loop   }              END;


 (temporary debugging check


 FOR CPrm:=1 TO Permutations DO                                             BEGIN
   writeln;
   FOR Sense := Positive DOWNTO Negative DO
    FOR Number := Singular TO Plural DO                                 BEGIN
     FOR Cphr:= 1 to Descendants - 1 DO
        S := Ptr[Calculate_Order(CPrm,Cphr,P_Table)]^.Role[number];
        Write(S,' ');
        IF Alt_sentences[CPrm,Cphr].Main = Yes
        THEN
           S:= Alt_sentences[CPrm,Cphr].Alt_phrases[Sense, Number]
        ELSE
           S:= Alt_sentences[CPrm,Cphr].Case_phrase;
        write(S,' ');
        S:= Ptr[ Calculate_Order(CPrm,Descendants,P_table)]^.Role[number];
                                                                       END END;    }
        writeln(S);                          {WITH  ptr[1]^   }             END;

 ranf:=ranf+1;

 writeln('Relation ', ranf-1:3, ' now added to dictionary.');
 END(of WITH RD)
 END; (of Create_Relation)
 .
```