ELFS: ENGLISH LANGUAGE FROM SQL

by

Stephen C. Kloster

B.A., Luther College, 1965

M.A., University of California, 1967

Ph.D., University of Iowa, 1971

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

SIMON FRASER UNIVERSITY

September 1985

ISBN 0-315-30778-1

APPROVAL

Name: Steve Kloster

Degree: Master of Science

Title of thesis: ELFS: English Language From SQL

Examining Committee:

Chairman: Dr. Tiko Kameda

Dr. Wo-Shun Luk
Senior Supervisor

Dr. Nick Cercone

(in absentia)
Dr. Veronica Dahl

Dr. James P. Delgrande
External Examiner
School of Computing Science
Simon Fraser University

29 August 1985
Date Approved

Title of Thesis/Project/Extended Essay

ELFS: English Language From SQL

Author:

(signature)

Steve Kloster

(name)

16 September 1985

(date)

ABSTRACT

We describe a system which, when given a query in a SQL-like relational database language, will display its meaning in clear, unambiguous natural language. The translation mechanism is independent of the application domain. The system has direct applications in the design of computer-based SQL tutorial systems and program debugging systems. The research results obtained in the thesis will also be useful in query optimization and the design of a front-end which will be more user-friendly than SQL.

ACKNOWLEDGEMENTS

I would like to thank Wo-Shun Luk for all of the assistance he has given me.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

This thesis describes a system named ELFS (English Language From SQL), which is capable of constructing an English translation of an SQL-like query. Given a query written in SQL, an English sentence will be produced which is equivalent to the query. Our main concern is to unravel the obscure, hard-to-understand structure of an SQL query imposed by the current SQL syntax. The basic approach adopted in this research is to analyse all SQL queries up to three levels deep, and then classify them into different types of queries.

The ELFS system has two major components: (i) the Query Transformer (QT) and (ii) the Natural Language Generator (NLG). The Query Transformer transforms complicated queries into pseudo-SQL queries. Simple queries will be left alone. In the NLG phase, the application-sensitive portions of the query, e.g., attribute names, relation names and constant values which have been left unchanged in the first phase, will be interpreted. Tables, similar to those found in [Codd78], are supplied by the users which contain phrases to express the association of two attributes in a relation. An English sentence that is equivalent to the query is then generated from the pseudo-SQL query with the help of these user-supplied tables.

SQL is one of the current database languages. A database language is an integral part of a database management system (DBMS) that provides the user with an interface to the system's internal functions. Some database languages are high-level nonprocedural query languages which allow nonprogramming users to express their database processing requirements in English-like queries without specifying how the data is to be retreived. Among all mainstream database models, the relation model is most amenable to the use of a query language as a database language. SQL (Structured Query Language) is perhaps the most popular query language for relational database systems. Two well-known relational database systems, IBM's SQL/DS and ORACLE Corp's ORACLE, employ SQL exclusively, either in stand-alone mode or embedded in a procedural language, to provide data definition, manipulation, and control facilities. Many DBMS's designed for microcomputers use SQL too. Although SQL is more than a query language for database accessing, the focus in this thesis is the data retrieval portion of SQL.

Versions of SQL have been in existence since the early 70's under the the name SEQUEL. The development of SEQUEL, in turn, is an evolutionary refinement of DSL ALPHA, a predicate calculus-based query language proposed by Codd as part of the relational model [Codd70]. In between ALPHA and SEQUEL was SQUARE, a query language which eliminated the need for quantifiers and bound variables required by ALPHA. To improve on SQUARE, SEQUEL replaced the concise mathematical notation of

SQUARE with a block-structured English keyword syntax. SQL is, by and large, just SEQUEL minus a few functions.

Although we expect the reader to be somewhat familiar with the SQL language, we explain, via a simple example, the main features of SQL which are relevant to the subject matter of this thesis. This example, like most of other examples cited below, is based on the familiar Part-Supplier-Project database schema:

PART(<u>P#</u>,PNAME,COLOR,WEIGHT,PCITY)

SUPPLIER(<u>S#</u>,SNAME,STATUS,SCITY)

PROJECT(<u>J#</u>,JNAME,JCITY)

SHIPMENT(<u>P#</u>,<u>S#</u>,<u>J#</u>,QTY)

If a 4-tuple (P#,S#,J#,QTY) belongs to the relation SHIPMENT, it means that part P# was sent by supplier S# to project J# in the quantity QTY. Attributes underlined are primary keys or parts of primary keys. In addition, PNAME, SNAME, and JNAME are candidate keys. All keys are unique by definition.

<u>Example 1</u>:

```
SELECT   PNAME
FROM     PART
WHERE    P# IN
        (SELECT   P#
         FROM     SHIPMENT
         WHERE    J#='J2')
```

This query consists of two query blocks. A query block has three clauses: SELECT-clause, FROM-clause and WHERE-clause. The first two clauses identify the attributes and the relation to be retrieved. The WHERE-clause is actually a predicate which, when evaluated against a given tuple of the relation in the

FROM-clause, yields either a 'TRUE' or 'FALSE' value. Recursively, a predicate can contain many predicates connected by logical connectives, i.e., 'AND' and 'OR'. In SQL, a predicate can contain another SELECT-FROM-WHERE block. We call this type of predicate a complex predicate. A predicate which is free of any nested SELECT-FROM-WHERE blocks is a simple predicate. In the above example, the last line contains a simple predicate. The predicate in the first WHERE-clause is a complex predicate, as it contains another query block. We call this query block a second level query block. Each query block returns a set of values. Blocks may be nested to arbitrary depth. We use the terms 'query block' and 'query (or subquery)' interchangeably. The query itself can be considered as a query block at the first level. The query block at the second or lower level can be considered as a subquery.

We adopt the version of SQL syntax as stated in Bradley [Brad82]. This syntax is a simplified version of SEQUEL II syntax [Cham76], for the purpose of illustrating the essential features of SQL. While it retains the structural complexity of SEQUEL II, it ignores minor details. For example, it allows the SELECT-FROM-WHERE blocks to be nested to arbitrary depth, but it does not specify, as SEQUEL II syntax does, the types and the formats of the constants it will accept. Since the current version of SQL is not identical to SEQUEL II, there are minor differences between IBM's SQL syntax and Bradley's own syntax. The most important difference is perhaps the 'CONTAIN' function, which was dropped during the modification of SEQUEL II. Because of this, it has been claimed (in [KimW82]), that the current version of SQL does not implement the division operation. However, as will be shown below, the 'CONTAIN' function may be emulated using the 'EXIST' and 'NOT EXIST' functions available in SQL. On the other hand, the 'EXIST' function, which is not found in Bradley's syntax, can be emulated by using 'DOES NOT CONTAIN' and the null set. Below we present the SQL Syntax according to [Brad82]:

```
SELECT [UNIQUE] attribute-1, attribute-2, ...
FROM   relation-1 [label-1], relation-2 [label-2], ...
     [WHERE requirement]
     [GROUP BY (attribute) HAVING requirement]
```

1. The requirement clause has the form
   1.1. requirement {AND, OR} requirement
   1.2. logical-relation

2. The logical-relation clause has the form
   2.1. attribute (¬=, =, >, <, ¬>,¬<) {constant, attribute}
   2.2. attribute {IN, NOT IN} {SQL-expression, set, SET (attribute)}
   2.3. {SQL-expression, set, SET(attribute)}, {CONTAINS, DOES NOT CONTAIN, ¬=, =} {(SQL-expression), set, SET(attribute)}

Consider the following scenario for retrieval of information by the user. The user has a request for information from the database. This request either exists in the user's mind, or is physically recorded on some medium. A query, in our case an SQL query, is formulated based on the given request. This query is input to the database system which subsequently produces the answer to the query. There are two translations involved in this process: from request to query, and then from query to answer. We assume throughout that the latter translation is always correct, i.e., the software/hardware system is bug-free. We explain below how the answer to an SQL query is derived. Of interest to us is the often imperfect process of query formulation.

We need a precise notion of incorrect translation from a request to a query. Let us first define query equivalence. Two queries are the same if they retrieve the same answer from every possible database that the database definition (i.e., database schema) permits. Conceptually, a request for information by the user can be treated as a (virtual) query. An SQL query constructed for a request for information is incorrect if there is at least one valid database that will yield an answer for that query which does not match the request.

We now describe how the answer to an SQL query is obtained strictly according to the syntax of SQL. We use Example 1 to illustrate this process. For each tuple of PART, the

WHERE-clause must yield a 'TRUE' value in order that the value of PNAME for that tuple be included in the answer set. To process the WHERE-clause, the query block at the second level must first be processed to yield a set of P#'s. To do so, each tuple of the relation SHIPMENT is retrieved to examine whether the WHERE-clause J#='J2' is satisfied. If that tuple has a value of J2 for the attribute J#, the P# of the tuple will be included in the answer set for the query block. When all the tuples of SHIPMENT have been examined, the answer of the query block is obtained. Then this answer is searched to determine whether the P# of the PART tuple is found there. When all the tuples of PART have been processed in this way, we have a set of PNAME values as the answer of the query. This answer will represent all the names of the parts that are used for project J2.

Queries such as Example 1 can be very easily understood by someone with some exposure to the SQL language. However, there are other SQL queries that are not so English-like. Many of these queries are useful in that they represent non-trivial requests for information from the database, but they are not easily accessible because of their hard-to-comprehend formats. Paradoxically, the evolution from DSL ALPHA to SQL was driven by the rationale that the database language should cater to users in the non-programming community. However, at least as far as SQL is concerned, the gain in ease of use may come at the expense (albeit a small one) of reduced power in retrieval. There is a certain limit to how far software designers can push for user-friendliness. Several human factors studies reveal that everyday English may not be the ideal way to communicate with computers [Shne78]. The universal quantifier is present in ALPHA, but not in SEQUEL (SQL). It is debatable whether the elimination of some mathematical (or logical) notation will help the user to formulate the query. We shall illustrate this point by considering the following request: "find the projects located in those cities which manufacture only red parts". One can express the query in ALPHA as follows:

```
    retrieve J# where V P (IF PCITY=JCITY THEN PART.COLOR=RED)
```

Note that we modify the syntax slightly for readability (i.e. IF A THEN B is to be interpreted as ¬A OR B). The equivalent query in SQL will be:

Example 2:

```
SELECT  J#
FROM    PROJECT
WHERE   JCITY NOT IN
       (SELECT  PCITY
        FROM    PART
        WHERE   PART.COLOR¬='RED')
```

The universal quantifier in the ALPHA query is disguised under the form ('NOT' '¬') in SQL. It is hard to understand this query without knowing how this form of 'double negation' is transformed into a universal quantifier. What may be more confusing is the fact that such a transformation is not always valid. Consider another request: find the suppliers which supply only red parts. At first glance, it seems the SQL query can be written as follows:

Example 3:

```
SELECT   S#
FROM     SHIPMENT
WHERE    P# NOT IN
        (SELECT  P#
         FROM    PART
         WHERE   PART.COLOR¬='RED')
```

However, this query will not retrieve the required information. Instead, it will get suppliers which supply at least one red part in addition to parts of any other colors. The correct query for this seemingly simple request takes on a rather formidable form:

Example 4:

```
SELECT S#
FROM   SHIPMENT  SHIPMENTX
WHERE  NOT EXIST
      (SELECT P#
       FROM   PART
       WHERE  PART.COLOR¬='RED')
       AND    P# IN
             (SELECT  P#
              FROM    SHIPMENT
              WHERE   SHIPMENT.S#=SHIPMENTX.S#)
```

There are actually two features in this query contributing to its opaqueness. The first is 'double negation', i.e., "NOT EXIST" and "PART.COLOR¬='RED'". The second has to do with the interblock reference 'SHIPMENT.S# = SHIPMENTX.S#'.

Senior undergraduate computer science majors in an introductory database course find that it is difficult to understand queries like the ones above, and even more difficult to construct them. Moreover, the students lack confidence in the correctness of the queries they construct themselves, since a slight displacement of a keyword may change the query to an inequivalent one. For example, it has been pointed out in [KimW82] that the following two queries are not equivalent:

Example 5:

```
a)    SELECT   SNAME                      b)     SELECT   SNAME
      FROM     SUPPLIER                          FROM     SUPPLIER
      WHERE    S# NOT IN                         WHERE    S# IN
              (SELECT    S#                              (SELECT   S#
               FROM      SHIPMENT                         FROM     SHIPMENT
               WHERE     P#='P1')                         WHERE    P#¬='P1').
```

Query a) will retrieve the names of suppliers which do not supply part P1, whereas query b) will retrieve the names of those suppliers which supply some part other than P1 (but may supply P1 as well).

This thesis is organized as follows. Chapter 2 contains an overview of ELFS with emphasis on the two important modules of the system, i.e., the Query Transformer (QT) and the Natural Language Generator (NLG). Chapters 3 and 4 are devoted to QT and NLG respectively. Chapter 5 describes the implementation of the ELFS program. In chapter 6 we outline some applications of our research results presented here. We give the conclusions and prospects for the future in chapter 7.

CHAPTER II

ELFS (ENGLISH LANGUAGE FROM SQL)

We have implemented a system, called ELFS, which is capable of producing an English sentence equivalent to a given SQL query. This system has two major components: (i) Query Transformer (QT) and (ii) Natural Language Generator (NLG). The input into ELFS is an SQL query formulated according to the SQL syntax as specified above.

To describe the division of labor between QT and NLG, let us analyse the contents of an SQL query. To understand an SQL query, we need two types of knowledge: the structure and the context of the query. It is presumed that the two are independent of each other and thus can be handled by QT and NLG respectively.

The structure of a query is independent of the attributes and relations which may vary from one application to another, although it is necessary for QT to know whether an attribute is a key or not. Basically, given the 'shell' of the query, QT determines whether the structure of the query is simple enough that NLG is capable of interpreting it and producing an English sentence equivalent to the query. If, in QT's judgement, the structure is too complicated, QT will transform it into a pseudo-query and pass it onto NLG. We define extended SQL to contain these pseudo-queries in addition to usual SQL queries.

The task of NLG is to mechanically translate the output of QT in an English sentence(s). Example 1 illustrates the type of query that QT will pass onto NLG without transformation. The output of NLG will be: "Select names of the parts which are shipped to project J2." If the WHERE-clause in the query block at the second level were instead to read: S#='S2', then the phrase "shipped to project J2" would be changed to "supplied by supplier S2". These phrases are constructed on the basis of the knowledge of the application which must be provided to NLG in some appropriate format.

The majority of queries are, of course, not so simple. Translating them literally without transformation will result in 'bad' English, or worse, generate misleading sentences. Consider Example 3. The query can be translated without transformation to read: "Select the S#-values of suppliers who ship parts not belonging to the set of parts which are not red." This is rather unclear. It certainly requires more effort to understand than the following: "Select the S#-values of suppliers who ship at least one red part(s)." There is a more important reason why literal translation would not work for some queries. Example 2 is a case in point. This query can be translated without transformation to read: "Select the J#-values of projects not located in those cities which do not produce red parts". The output of this translation is definitely confusing. If one reads this sentence carefully, he will arrive at a <u>wrong</u> interpretation of the query!! The output indicates that projects located in cities which produce no red part will be excluded from the answer. In other words, projects located in cities which produce _at_ _least_ one red part and possibly _parts_ _of_ _other_ _colors_ will be included in the answer. In fact, the true interpretation is quite different: "Select J#-values of projects located in those cities which produce only red parts."

In the following chapters, we discuss in detail the functions performed by QT and NLG respectively.

CHAPTER III

QUERY TRANSFORMER (QT)

In this chapter, we identify precisely what types of queries will be processed by QT, and then we show how these queries will be transformed. Proofs will be provided to show that query transformation does not alter the answer of the original query.

We have been able to identify a few features or functions of SQL that render a query difficult to understand or open to misinterpretation.

(i) Negation: A predicate in a WHERE-clause of a query block is said to be in negative form if it contains a 'NOT' or its equivalents (e.g. '¬'). As noted in Example 5, negative predicates are easily misinterpreted. Nested negative predicates, such as 'double negation' and 'triple negation', are much worse. We discovered that the confusion has something to do with the uniqueness (or lack thereof) of the attribute(s) in the SELECT clause of the query block with the negative predicate. We call this attribute(s) the link attribute(s). In Example 2, the attribute PCITY in the SELECT-clause of the query block at second level is not a unique attribute; i.e., PCITY is not a key in the relation PART. In contrast, the corresponding attribute P# in Example 3 is the key of the relation PART. That explains why two seemingly similar queries may not be interpreted in the same way.

(ii) Interblock Referencing: This refers to the situation where the WHERE-clause of a query block refers to attribute(s) of a relation in a FROM-clause outside the query block. Most of the predicates beginning with 'EXIST' or 'NOT EXIST' will contain interblock references. It is particularly difficult to understand the query when the relations in the FROM-clauses of two nested query blocks are essentially the same, such as the clause in Example 4, 'SHIPMENT.S#=SHIPMENTX.S#', where SHIPMENTX is an alias of SHIPMENT.

Guided by the above analysis, we now classify queries into various categories and develop a transformation scheme for each category of queries. Before we do so however, we have to define the scope of the queries we are prepared to deal with. In this thesis, we restrict ourselves to those queries with up to three levels of query blocks. Some support for this restriction can be found in the fact no query in Date's book [Date81] has more than three levels. We hope the reader will understand the essence of our interpretation scheme and share our belief that there will not be any major conceptual difficulties when we relax this restriction, although a greater number of classes of queries would have to be handled.

We present an outline form of the master plan for QT which shows how the queries are classified and what transformation method is used for each class. Part of the classification depends on where the negations occur. For example, we use the notation '(+,-)' to denote a nested query in which the outer block is positive, and the inner block is negative.

1. No 'EXIST' occurs in the query
    a. No interblock references: NtoP Rule (theorems 1 and 2)
    b. Interblock references
        1) Adjacent blocks
            a) Different relations: NtoP Rule
            b) Identical relations
                i) (+,+): always true
                ii) (-,+): always false
                iii)(+,-): 'AT LEAST ONE'
                iv) (-,-): 'NUMBER = 1'
        2) Non-adjacent blocks: analysis by case
2. 'EXIST' occurs
    a. Double negations: theorems 3 and 4
    b. Single negations or no negations: translate directly

The primary division classifies queries into two large groups: the group of queries without any occurence of 'EXIST' (or 'NOT EXIST'), and the other group in which there is such an occurence.

Queries without 'EXIST'

NtoP Rule

Fundamental to the processing of the query at this stage is a transformation rule which converts a negative predicate into a positive one by 'passing on' the negation to the predicate one level below. Hence we call this transformation rule the NtoP Rule. This transformation rule is justified by the following two theorems.

These theorems concern two different cases. The first case occurs when the link attribute is unique, and the second occurs when it is not unique. In the unique case, the negation can be 'passed on' in a simple way. This is done in theorem 1. The non-unique case is not so simple, and leads to the introduction of the quantifier 'FOR ALL'. Example 3 illustrates the unique case, and Example 2 illustrates the non-unique case.

Theorem 1: If the link attribute L is unique in the relation R, and R has no null values, then all instantiations of the forms a) and b) below are equivalent as WHERE clauses in any query:

```
 a)  L NOT IN                      b) L IN
     (SELECT L                        (SELECT L
      FROM   R                         FROM   R
      WHERE  COND)                     WHERE  ¬COND)
```

Proof: Let L1,L2,...,Ln be the attributes of R and suppose that L is the first attribute, L1. Let R(L1,L2,...,Ln) denote the proposition which asserts that the n-tuple (L1,L2,...,Ln) belongs the relation R. The proposition is true when the n-tuple belongs to the relation, and false when the n-tuple does not. Clause a) asserts that L is not in the set S, where

S={L1: exists L2,...,Ln (R(L1,L2,...,Ln)&COND)}.

This is equivalent to asserting that L is in the complement of S. Denoting the complement by COMP(S), we have COMP(S)

={L1: for all L2,...,Ln (R(L1,...,Ln)->¬COND)}.

But now since L1 is unique in R, all the values for L2,...,Ln are determined by L1, so we don't need the universal quantifier. COMP(S) is

{L1: (R(L1,...,Ln)->¬COND)}.

Since there are no null values, for each value of L1, there exist L2,...,Ln such that R(L1,L2,...,Ln). Thus COMP(S) is

{L1: ¬COND)}.

This set is the one described by clause b), completing the proof.

Theorem 2: If the link attribute NU is not unique in the relation R, and R has no null values, then all instantiations of the forms a) and b) below are equivalent as WHERE clauses in any query:

```
 a)  NU NOT IN                    b) NU IN
       (SELECT NU                     (SELECT NU
        FROM   R                       FROM    R
        WHERE  COND)                   WHERE   FOR ALL X IN RX
                                      (RX.NU=R.NU ->¬COND))
```

Proof: Let L1,L2,...,Ln be the attributes of R, and for the sake of argument, suppose that NU is the first attribute, L1. (This will not affect the nature of the proof.) Clause a) asserts that NU is not in the set S where

S={L1: exists L2,...,Ln (R(L1,L2,...,Ln)&COND)}.

This is equivalent to asserting that NU is in the complement of S. Denoting the complement by COMP(S), we have COMP(S)

={L1: for all L2,...,Ln (¬(R(L1,...,Ln)&COND))}

={L1: for all L2,...,Ln (¬R(L1,...,Ln) or ¬COND)},

by De Morgan's law. Using the definition of implication, this can be written as

{L1: for all L2,...,Ln (R(L1,...,Ln)->¬COND)}.

To assert that NU is in this set is equivalent to saying

FOR ALL X IN RX (RX.NU=R.NU ->¬COND)}.

This set is the one described by clause b), thus completing the proof of theorem 2.

We have seen that form b) arises in a natural way from the proof. However, it does contain some information which is redundant. Form c) below is more compact.

```
c) NU IN
     (SELECT NU
      FROM   R
      WHERE  FOR ALL R ¬COND)
```

We extend SQL so that WHERE clauses of the form

FOR ALL R COND

are allowed. All queries of form c) will be included in this "extended SQL". This form is utilized to construct the translation of the query.

The universal quantifier "FOR ALL X" in form b) ranges over the set of all n-tuples of the relation R. Note that RX is just another copy of the relation R. The WHERE clause in b) asserts that if we take any n-tuple X of RX, if the NU-attribute has the specified value R.NU, then ¬COND holds. If there are any interblock references in COND, they are treated as constants.

Now we describe how the NtoP Rule is applied to the negation of a predicate. The first step is to remove the 'NOT' from the predicate, and then negate the predicate one level below. If the link attribute in the subquery one level below is non-unique, add the keyword 'FOR ALL' before the predicate at that level. Typically, the transformation rule is applied to the query repeatedly from the first level until the only possible negative predicate in the entire query is at the bottom level. No transformation is needed for a positive predicate, except that if the attribute in a clause of the predicate can have multiple values associated with one value of the link attribute, we must add the phrase 'AT LEAST ONE' before the clause.

Let us give some examples showing how the NtoP Rule is applied. Recall that in Example 2 the WHERE clause is

```
JCITY NOT IN
     (SELECT PCITY
      FROM   PART
      WHERE  PART.COLOR¬='RED')
```

By theorem 2, this can be transformed to

```
   JCITY IN
        (SELECT PCITY
         FROM   PART
         WHERE  FOR ALL PARTS PART.COLOR='RED')
```

In the relation PART, the attributes P#, PNAME, COLOR, and WEIGHT are considered as variables which are free to vary over the tuples of PARTS. The effect of the condition 'PARTX.PCITY=PART.PCITY' in form b) is to fix the value of PCITY. One translation of this clause would be "cities having the property that for all parts p, if p is made in the city, then p is colored red". The advantage of translations like the above is that they can be used for a large class of queries. The disadvantage is that the output is not as clear as it could be. A simpler translation is "cities making only red parts". In the next example, the WHERE clause is the conjunction of two conditions.

Example 6:

```
   PCITY NOT IN
        (SELECT PCITY
         FROM   PART
         WHERE  PART.COLOR¬='RED'
         AND    PART.WEIGHT=20)
```

By theorem 2, this is equivalent to

```
   PCITY IN
        (SELECT PCITY
         FROM   PART
         WHERE  FOR ALL PARTS
          (PART.COLOR='RED' OR PART.WEIGHT¬=20))
```

There is a choice of translations in this case. One possibility is "cities having the property that for all parts p, if p is made in the city, then p is colored red or p does not weigh 20 lbs." Another possibility is "cities having the property that all parts made in the city weighing 20 lbs. are red."

The WHERE clause above is a conjunction of the form ¬C1 AND C2. For theorem 2 we need the negation of this, which is C1 OR ¬C2. This can be expressed as C2 -> C1, and leads us to the second translation above. To decide which translation is easier to understand could be a topic for further research. It seems to us, however, that reducing the number of negations to the minimum produces the most understandable result. For instance, if the WHERE clause has the form

¬C1 & ¬C2 & ... & ¬Ci & Ci+1 & ... & Cn

and there are no negations in the C's, then the negation of the clause could be expressed as

Ci+1 & ... & Cn -> C1 OR C2 OR ... OR Ci.

This form has no negations, and we expect that it would be the most understandable form which is logically equivalent to the original.

For the rest of this thesis, predicates with negative as well as positive clauses are considered as negative predicates and they will be processed in the manner as described above.

Queries without 'EXIST' and without Interblock References

We are now ready to describe the algorithm to transform queries in the class mentioned in the heading. For a query block with positive predicates, the phrase 'AT-LEAST-ONE' is inserted before the predicate if the link attribute is not unique. For query blocks with negative predicates, application of the NtoP Rule alone is sufficient. A 3-level query with an even number of negative predicates will become a positive query. Otherwise, the transformation will result in a negative predicate at the bottom level. Let us denote eight possible combinations of positive/negative predicates according to the level it occurs, by (+,+,+), (+,+,-), ..., and (-,-,-). The NLG is certainly capable of translating cases such as (-,+,+) or (+,-,+) without transformation at this stage. The output might perhaps be improved by using NtoP Rule. For simplicity, we adopt the method of "pushing" all negation to the lowest level possible.

Queries without 'EXIST' but with Interblock References

First, we consider the type of interblock references where the predicate of a subquery refers to an attribute of a relation in the query block immediately above the subquery. A typical query is shown below:

```
SELECT  *
FROM    R1
WHERE   L12 {NOT} IN
        SELECT L2
        FROM   R2
        WHERE  L23 {¬}= R1.L13
```

The items in curly brackets may or may not be present. This query implies a match (or mismatch) of two relationships: L12-L2 in R1 and L23-L13 in R2. Should we treat these relationships as identical, even when they are in different contexts (i.e. different relations)? The answer to this question involves a much larger issue: the universal relation assumption (URS) and the controveries surrounding it ([Atze82] and [Ullm82]). We have avoided this issue so far and now explain the issue in the context of this research.

There are several versions of URS, with subtle differences among them. Two of them concern us here. One of them presumes the uniqueness of the meaning of the attribute within the entire schema. In other words, the meaning of the attribute is identical in whatever relation it appears. The other version presumes the uniqueness of relationship of each pair of attributes within the entire schema. Our position concerning these assumptions is that the user is the one to decide whether any of these assumptions is valid and he can communicate his decision through the tables for attribute associations which will be used by NLG to produce a sentence. For simplicity, we have adopted the assumption of uniqueness of meaning of an attribute within the entire database schema. However, it can be seen that the theorems do not rely on this assumption, so are still valid without it. In our database, PCITY, JCITY and SCITY all mean cities and as such are used as link attributes in subqueries. It seems very awkward to have to refer to PCITY as "cities where parts are produced" all the time.

On the other hand, we assume the relationships are different if the relations are different. Therefore there will be no extra meaning beyond the simple value matching of the two relationships. For example, let R1(EMPLOYEE, EMPLOYEE) and R2(EMPLOYEE,EMPLOYEE) be the two relations in the above example. We shall assume the relationship EMPLOYEE-EMPLOYEE in R1 to be different from the relationship EMPLOYEE-EMPLOYEE in R2. For instance, the former could be father-son and the latter foreman-worker. The query then requests information about foremen who have their sons working under them. Thus, for this type of query, we shall first apply the NtoP Rule and then transform it into another type of pseudo-query, such as the following:

```
SELECT   *
FROM     R1  R2
WHERE    FOR  R1.L12 = R2.L2
         (FOR ALL or AT-LEAST-ONE) R2: R2.L23 {¬}= R1.L13
```

If R1 and R2 are identical, then the two relationships will be treated as identical relationships, and therefore a different tansformation method must be used. Instead of value matching, each of the four possible cases, i.e. (+,+), (+,-), (-,+), and (-,-) is to be interpreted differently. The (+,+) and (-,+)

cases are trivial, the predicate of the former being always true and that of the latter being always false. The (+,-) case is meant to retrieve all tuples where the value of L12 (or L2) is associated with at least two different values of L12 (or L23) in different tuples. Consider the following example:

Example 7:

```
SELECT   *
FROM     SHIPMENT SHIPMENTX
WHERE    S# IS IN
         SELECT  S#
         FROM    SHIPMENT
         WHERE   J# ¬= SHIPMENTX.J#
```

The above query will retrieve information about suppliers who supply parts to at least two different projects. The predicate at the bottom will be transformed into 'AT-LEAST-TWO-DIFFERENT J#'.

The (-,-) case of the above example after the NtoP transformation is shown as follows:

Example 8:

```
SELECT   *
FROM     SHIPMENT SHIPMENTX
WHERE    S# IS IN
         SELECT  S#
         FROM    SHIPMENT
         WHERE   FOR ALL SHIPMENT: J#= SHIPMENTX.J#
```

The predicate at the bottom asserts that all projects to which the supplier ships have one project number. Thus we can replace the predicate with the predicate 'NUMBER(J#) = 1'.

What remains to be handled in this category of queries without the 'EXIST' keyword is the class of queries where interblock references are not between two neighbouring query blocks. A typical query is shown below:

```
SELECT   *
FROM     R1
WHERE    L12 {NOT} IN
         SELECT   L2
         FROM     R2
         WHERE    L23 {NOT} IN
                  SELECT   L3
                  FROM     R3
                  WHERE    L31 {¬}= R3.L13
```

The approach to transform this type of query is almost identical to the one for the type of queries with two neighbouring query blocks referring to each other. Hence, we shall not describe it here. It suffices to say that the pseudo-queries after transformations will have no interblock references, just as other pseudo-queries we have created.

<u>Queries with 'EXIST'</u>

We shall first show that all occurrences of'EXIST' can be eliminated by transforming the query to one which uses 'CONTAINS'. This transformation is valid for queries of arbitrary depth. If a query has n occurences of 'EXIST', the transformation will be applied n times, and the result will not contain any 'EXIST's.

This general scheme has the drawback that the resultant query may not have a very natural direct translation. (An example is given below.) Theorem 3 provides us with an improvement over the general scheme. In theorem 3, we show that a pair of 'NOT EXISTS' can be transformed to a single 'CONTAINS' which provides a more natural interpretation.

If 'EXIST' occurs in a WHERE clause, it must occur either positively or negatively. The positive form may be written as follows:

```
 EXIST
(SELECT *
 FROM   R
 WHERE  COND)
```

This can be transformed to

```
   EMPTYSET
DOES NOT CONTAIN
  (SELECT *
   FROM   R
   WHERE  COND)
```

Here EMPTYSET denotes the empty set. It can be defined in any database. One definition would be

```
(SELECT *
 FROM   R
 WHERE  FCOND)
```

where FCOND is always false.

The negative form of 'EXIST' may be written as follows:

```
 NOT EXIST
(SELECT *
 FROM   R
 WHERE  COND)
```

This can be transformed to

```
    EMPTYSET
CONTAINS
   (SELECT *
    FROM   R
    WHERE  COND)
```

In the above discussion, the predicate COND was allowed to contain nested queries. Let us define a simple predicate to be one which does not contain any nested query. In this special case, a simpler transformation can be given. If COND is a simple predicate, consider any query of the form below:

```
SELECT A
FROM   R1
WHERE  EXIST
      (SELECT *
       FROM   R2
       WHERE  COND)
```

If COND does not have any reference to any attribute of R1, the query is superfluous because either all or none of A-values of R1 will be retrieved. On the other hand, if COND takes the form: 'R2.B = R1.B', then the query can be reduced to one with no 'EXIST':

```
SELECT A
FROM   R1
WHERE  R1.B=R2.B
```

We give an example of a query which contains several 'EXIST's. (This is example 7.26 in [Date81])

Example 9:

```
SELECT J#
FROM   SHIPMENT SHIPMENTX
WHERE  NOT EXIST
           (SELECT *
            FROM   SHIPMENT SHIPMENTY
            WHERE  EXIST
                   (SELECT *
                    FROM   SHIPMENT
                    WHERE  S#='S1'
                    AND    P#=SHIPMENTY.P#)
            AND NOT EXIST
                   (SELECT *
                    FROM   SHIPMENT
                    WHERE  S#='S1'
                    AND    P#=SHIPMENTY.P#
                    AND    J#=SHIPMENTX.J#))
```

If we were to transform this using the EMPTYSET transformations, we would end up with the following:

```
SELECT J#
FROM   SHIPMENT SHIPMENTX
WHERE       EMPTYSET
            CONTAINS
           (SELECT *
            FROM   SHIPMENT SHIPMENTY
            WHERE  EXIST
                   (SELECT *
                    FROM   SHIPMENT
                    WHERE  S#='S1'
                    AND    P#=SHIPMENTY.P#)
            AND     EMPTYSET
                    CONTAINS
                   (SELECT *
                    FROM   SHIPMENT
                    WHERE  S#='S1'
                    AND    P#=SHIPMENTY.P#
                    AND    J#=SHIPMENTX.J#))
```

This query is still very hard to understand.

Queries like Example 9 arise when one wants to express queries which involve universal quantification in a version of

SQL which does not have 'CONTAINS'. Then one must resort to using a double 'NOT EXIST'. Theorem 3 shows that those queries which use a double 'NOT EXIST' to express universal quantification can be mechanically transformed into queries using 'CONTAINS'.

<u>Theorem 3</u>: All instantiations of forms a) and b) below are equivalent as WHERE clauses in any query.

```
a) NOT EXIST                        b) (SELECT A
     (SELECT *                           FROM   R2
      FROM   R1 R1Y                      WHERE  COND2)
      WHERE  COND1                      CONTAINS
      AND NOT EXIST                     (SELECT A
         (SELECT *                       FROM   R1Y
          FROM   R2                      WHERE  COND1)
          WHERE  COND2
          AND    R2.A=R1Y.A))
```

<u>Proof</u>: We assume that the clause R2.A=R1Y.A is the only clause joining the two blocks (i.e., joining R2 to R1Y.) (If there were other clauses, we could treat A as a tuple, and the proof would be generalized.) Notice that in b) COND2 comes before COND1. Also, the clause joining relations R1 and R2 is not needed in b).

Clause a) asserts the following:

NOT EXIST A,Y2,...,Ym {R1(A,Y2,...,Ym) AND COND1

AND NOT EXIST X2,...,Xn (R2(A,X2,...,Xn) AND COND2)}.

We assume that R1 is an m-ary relation and R2 is an n-ary relation, and that the joining attribute A is the first attribute in both relations. We indicate the fact that the relations are joined on this attribute by using the variable A in both relations. The above condition is equivalent to

FOR ALL A,Y2,...,Ym ¬{R1(A,Y2,...,Ym) AND COND1

AND NOT EXIST X2,...,Xn (R2(A,X2,...,Xn) AND COND2)}.

By De Morgan's law, this can be written

FOR ALL A,Y2,...,Ym {¬(R1(A,Y2,...,Ym) AND COND1)

OR EXIST X2,...,Xn (R2(A,X2,...,Xn) AND COND2)}.

Equivalently,

FOR ALL A, {NOT EXIST Y2,...,Ym (R1(A,Y2,...,Ym) AND COND1)}

OR {EXIST X2,...,Xn (R2(A,X2,...,Xn) AND COND2)}.

Or,

FOR ALL A, {EXIST Y2,...,Ym (R1(A,Y2,...,Ym) AND COND1)}

-> {EXIST X2,...,Xn (R2(A,X2,...,Xn) AND COND2)}.

This last condition is exactly what clause b) asserts, and this completes the proof.

We now apply theorem 3 to Example 9. When matching this example to the form a), we see that R2 is 'SHIPMENT', R1 is 'SHIPMENT', and R1Y is 'SHIPMENTY'. Only one clause joins these relations, and that is 'P#=SHIPMENTY.P#'. Thus the attribute A is 'P#', and COND2 is "S#='S1' AND J#=SHIPMENTX.J#". Note that in the original query, the clauses which form COND2 may be separated by the join clause. After applying theorem 3, the query is transformed into the following query:

```
SELECT J#
FROM   SHIPMENT SHIPMENTX
WHERE (SELECT P#
       FROM   SHIPMENT
       WHERE  S#='S1'
       AND    J#=SHIPMENTX.J#)
      CONTAINS
      (SELECT P#
       FROM   SHIPMENT SHIPMENTY
       WHERE  EXIST
                (SELECT  *
                 FROM    SHIPMENT
                 WHERE   S#='S1'
                 AND     P#=SHIPMENTY.P#))
```

This query still contains one 'EXIST', but it occurs with a predicate that refers to the attribute P#. According to the analysis given just before the statement of Example 9, the entire predicate from 'EXIST' to the end of the query can be replaced by one single predicate, "S#='S1'". Using the transformation for this special case, we obtain

```
SELECT J#
FROM   SHIPMENT SHIPMENTX
WHERE (SELECT P#
       FROM   SHIPMENT
       WHERE  S#='S1'
       AND    J#=SHIPMENTX.J#)
      CONTAINS
      (SELECT P#
       FROM   SHIPMENT
       WHERE  S#='S1')
```

The subquery situated above the word 'CONTAINS' selects those parts supplied to project J# by S1. The subquery below the word 'CONTAINS' selects all the parts supplied by S1. This transformed query, after being passed on to the NLG, would be translated as follows:

"Select the J#-values of projects satisfying the following conditions: if part P# is supplied by S1, then S1 sent a shipment of part P# to project J#." The translation Date gives for this query is

"Get J# values for projects supplied by supplier S1 with all parts that supplier S1 supplies".

The next theorem is similar to Theorem 3, and is useful for queries like Example 4. It permits us to eliminate a double negation.

Theorem 4: All instantiations of the forms a) and b) below are equivalent as WHERE clauses in any query.

```
a) NOT EXIST
      (SELECT A
       FROM   R1
       WHERE  NOT COND1
       AND A IN
          (SELECT A
           FROM   R2
           WHERE  COND2)
```

```
b) (SELECT A
    FROM   R1
    WHERE  COND1)
    CONTAINS
   (SELECT A
    FROM   R2
    WHERE  COND2)
```

Proof: Analogous to theorem 3.

When we apply Theorem 4 to Example 4, we obtain

```
SELECT  S#
FROM    SHIPMENT  SHIPMENTX
WHERE
       (SELECT  P#
        FROM    PART
        WHERE   PART.COLOR='RED')
      CONTAINS
       (SELECT  P#
        FROM    SHIPMENT
        WHERE   SHIPMENT.S#=SHIPMENTX.S#)
```

The translation of this query is "Select the S#-values of suppliers who supply only red parts."

CHAPTER IV

NATURAL LANGUAGE GENERATOR (NLG)

The method used by NLG is based on the tables approach originating from Codd's work on the RENDEZVOUS project. The aim of Codd's project was to allow users to use English to interrogate a database. The initial input was an English sentence, which was translated internally into a query in DEDUCE. This was then translated back into English and displayed. The user could then either confirm that ~~it~~ captured his intention or request a correction. The translation back into English was called the generation step, and it is this step which is similar to our work. However, Codd did not develop his system to the point where it could handle universal quantification, the general use of negation, or the use of 'OR' and 'AND'. Our system will handle all of these features, and thus extends Codd's work. Some new ideas were introduced; for example, recursion was used ~~for~~ the 'OR' case. Codd's translation was from DEDUCE into English, while ours is from SQL into English. There are many database systems which use SQL, and our program could easily be adapted to be used with them.

Another difference between our system and Codd's is that Codd combined nouns and adjectives in the same table. Since a noun phrase may contain any number of adjectives, but may only have one noun, it is advantageous to keep the table for the adjectives distinct from the one for the nouns.

One of the advantages of the table method is that all of the domain specific information is located in a small number of tables. When changing from one domain to another, the tables can be changed in a straightforward manner. Below we consider a sample query and show how the table method is used. This example should make it clear how the domain specific information can be put into a table.

To discuss our ideas, let us consider the following example of an SQL query:

```
SELECT J#
FROM   SHIPMENT
WHERE  S#='S1'
```

Our translation of this query will be: "Select the J#-values of projects supplied by supplier S1." Our method of translation involves setting up tables for each of the relations. Here, for example, is the table for the 'SHIPMENT' relation:

SHIPMENT(A,B)

	B-attribute	English phrase
A=P#	*	supplied
	QTY	in a quantity
	S#	by
	J#	to
A=S#	*	who sent a shipment
	P#	of
	QTY	in a quantity
	J#	to
A=J#	*	supplied
	P#	with
	QTY	in a quantity
	S#	by

Here A refers to the attribute involved in the SELECT clause. The value of A determines which part of the table to use. If A=P#, use the first four rows, if A=S#, use the second four rows, and if A=J#, use the last four rows. (Notice that the English phrase depends mostly on the value of B, but does depend also on the value of A. Since it depends on two attributes, it is an example of a two-dimensional table.)

It is straightforward to generate the first part of the translation "Select the J#-values of projects". To obtain the remainder, we use the table. In the example, the SELECT clause contains J#. Thus A is J#, which means we must use the bottom four rows of the table. The right column contains the English phrase. The asterisk indicates that the corresponding phrase is always to be output. In our example, we get 'supplied'. Let COND denote the complete WHERE clause. In our case, COND is "SHIPMENT.S#='S1'". The table is scanned sequentially. Whenever one of the attributes in the left column occurs in COND, we output the phrase to the right. (This phrase needs to be completed, and this is done by another table.) In our case, P# and QTY do not occur in COND. However, the attribute S# does occur, so 'by' is output. The completion of this phrase, 'supplier S1', is provided by a table for the S relation.

As another illustration of the table method, let us consider the following query:

```
SELECT J#
FROM   SHIPMENT
WHERE  S#='S1' AND P#='P3'
```

Again, the SELECT clause contains J#, so we go to the bottom four rows of the table. We must use the phrase across from the asterisk, and so we get 'supplied'. This time COND is "S#='S1' AND P#='P3'". The attributes given in the remainder of the table are P#, QTY, and S#, in that order. We examine COND, looking for each attribute in turn. Since P# occurs in COND, output 'with'. Next we output 'parts P3', since that is the value given in COND. Then we go looking for 'QTY' in COND. It does not occur, so the phrase 'in a quantity' is not output. Finally, we get to the last line of the table, which concerns 'S#'. It does occur in COND, and we get 'by' as in the previous example. In addition, we also get 'supplier S1'. Putting everything together, the translation is: "Select the J#-values of projects supplied with parts P3 by supplier S1."

Tables can be extended so as to take care of single negations, but not double negations. When a double negation is encountered, we invoke the query transformer (QT). The query is transformed into an extended SQL form, which is passed on to NLG. NLG is able to translate this extended SQL form in a straightforward way by making use of the existing tables.

CHAPTER V

IMPLEMENTATION OF ELFS

One implementation of ELFS was done in PL/I, and one was done in Prolog. The Prolog program is given in the Appendix. First we describe exactly what inputs are allowed to the program. Then we proceed to describe the details of the algorithms used.

The program can handle queries of the following form:

1. Any level 1 query of the form below:

```
SELECT A
FROM   R
WHERE  COND
```

2. Any level 2 query of the form below:

```
SELECT A
FROM   R
WHERE  B {NOT} IN
      (SELECT B
       FROM   R2
       WHERE  COND2)
```

The curly brackets around the 'NOT' indicates that it may be either present or not present. Both single and double negations are allowed. Any number of 'AND's and 'OR's are allowed in the WHERE clauses. At present, interblock reference and 'EXIST' are not allowed. The 'CONTAINS' and 'GROUP-BY' constructs are not implemented.

The basic translation algorithm used by NLG is given below. Any level 2 query can be put in the same form as the level 1 query above if we allow COND to contain a query. We will allow this in the following discussion. Let LOGICOP be the major connective in COND. If LOGICOP='AND', or if COND has only one condition, we use this translation scheme:

1. "Select the"+NOUNS(A)+PREP(A)
2. TRANS(A,0,COND)
3. TABLES(R,0,COND,R(A)).

If LOGICOP='OR', (which means that COND can be written 'P1 OR P2'), we use the following alternate scheme:

1. "Select the"+NOUNS(A)+PREP(A)+NOUNS(R(A))
2. "satisfying the following conditions:"
3. "they are"+TRANS(A,0,P1)+TABLES(R,0,P1,R(A))
4. LOGICOP
5. "they are"+TRANS(A,0,P2)+TABLES(R,0,P2,R(A))

These algorithms use four subroutines: NOUNS, PREP, TRANS, and TABLES. TRANS invokes a fifth routine called MOD.

NOUNS simply translates the attribute A into the corresponding noun. Thus SNAME gives "names", SCITY gives "cities", and so on. This routine is domain-dependent.

PREP selects the proper preposition to use for the attribute A. For example, for SNAME, we should use "of", but we should not use "of" for SCITY. A translation like "Select the cities of suppliers who sent a shipment to London" is a bit awkward. A better translation is "Select the cities with suppliers who sent a shipment to London". This routine, like NOUNS, is domain-dependent.

The purpose of TRANS is to find a complete noun phrase. TRANS takes 3 arguments, A, NOT, and COND. NOT is a boolean variable which indicates whether or not negation is in effect. The value 0 indicates that no negation is involved. The translation schemes given above use the value 0 in the top level calls to TRANS and TABLES. TRANS is quite simple, as it contains only two steps:

1. NOUNS(R(A)).
2. MOD(A,NOT,COND).

The function R(A) finds the relation to which the attribute A belongs.

MOD is a routine which finds all the modifiers of A in the condition COND. Since a noun phrase may contain any number of modifiers, the MOD routine must have a recursive nature. Notice that MOD does not have to concern itself with finding the noun; that is the job that TRANS is responsible for. Although MOD must refer to some domain-dependent files, we have succeeded in keeping it domain-independent. It takes 3 parameters as input: A, NOT, and COND. MOD consists of the following 5 steps:

1. Check for the occurrence of 'FOR ALL' and process accordingly.

2. If COND includes an 'IN' form like

```
A {NOT2} IN
  (SELECT A
   FROM   R2
   WHERE  COND2),
```

```
then: a. call MOD(A,NOT+NOT2,COND2)
      b. If R2 is multi-dimensional,
         call TABLES(R2,NOT,COND2,R(A)).
```

3. If COND includes an equality, 'R.A = A-constant', output A-constant.

4. If COND has major connective LOGICOP,

```
then: a. call MOD(A,NOT,P1),
      b. output LOGICOP(NOT),
      c. call MOD(A,NOT,P2), and return.
```

We should output LOGICOP only if both MOD's return non-empty strings.

5. Find the table for the relation R(A) and go through the attribute list.
   a. Set B to the first attribute of the table.
   b. If there are occurences of R.B in COND, they should be of the form 'R.B OP X'.
      Call PRINTCOMP(OP,X).
   c. Move to next the next attribute B and go to step b above.

PRINTCOMP stands for "print comparison". For example, PRINTCOMP(<,30) will give us the English phrase "less than 30".

The TABLES routine performs the task of scanning the multi-dimensional tables of the database. (In our sample database, the only such table is the one for SHIPMENT.) TABLES takes four parameters: R, NOT, COND, and A. It has 3 steps.

```
1. If OR is the major connective, (i.e., COND=P1 OR P2),
   a. call TABLES(R,NOT,P1,R(A)),
   b. output OR(NOT),
   c. call TABLES(R,NOT,P2,R(A)), and return.

2. Set B to the first attribute in the table. (If ¬NOT is
   in effect, use an alternate table with negations.)

3. While B¬=' ' do
  a. If B='*', then put the English phrase in buffer
     called phrase1.
  b. If 'R.B' occurs in a predicate P in COND, and is
     the first such occurrence, output phrase1 and the
     phrase corresponding to B.
     If it is not the first occurrence, output only the
     phrase corresponding to B.
     Call TRANS(B,0,P). (To find any modifiers of B).
  c. If B=QTY, set the value of COMPCOL.
  d. Move to next line of table.
```

PRINTCOMP takes two parameters, OP, and B-constant. COMPCOL is a flag that is set by TABLES, and determines the phrase to be used in comparisons. The steps of PRINTCOMP are listed below:

1. If OP is '=', output B-constant.

2. If OP is '¬=', output "other than" B-constant.

3. Else use column COMPCOL

Let us give some complete examples to show how the routines of NLG work.

Example 10:

```
SELECT  S#
FROM    SHIPMENT
WHERE  (SHIPMENT.J#='J1'
     OR SHIPMENT.J#='J2')
AND     P# IN
       (SELECT P#
        FROM   PART
        WHERE  PART.COLOR='RED');
```

The translation for Example 10 comes out to be: "Select the S# values of suppliers who sent a shipment of parts colored red to project J1 or project J2."

Even though this is a level 2 query, we can put it into the general form given for a level 1 query by taking A to be S#, R to be SHIPMENT, and COND to be everything after the first "WHERE". Note that the major connective in COND is "AND", so we use the simpler translation scheme.

According to step 1, we must find NOUNS(S#) and PREP(S#). NOUNS(S#) is "S#-values", and PREP(S#) is "of". Thus the output from step 1 is the phrase "Select the S#-values of".

In step 2, we obtain TRANS(S#,COND). Now NOUNS(R(S#))= NOUNS(SUPPLIER)="suppliers". MOD(A,NOT,COND) turns out to be the empty phrase, so step 2 gives us "suppliers".

In step 3 we must find TABLES(SHIPMENT,NOT,COND,SUPPLIER), where COND is

```
             (SHIPMENT.J#='J1'
          OR SHIPMENT.J#='J2')
      AND    P# IN Q2
```

and the subquery Q2 is

```
            (SELECT P#
             FROM   PART
             WHERE  PART.COLOR='RED');
```

The table for the relation SHIPMENT was given in chapter 4. The rows corresponding to "SUPPLIER" are those in the middle third of the table. First we get the English phrase "who sent a shipment", by step 3a of the TABLES routine. Going down the table, we come to P#. We see that P# does occur in COND, so output "of". The phrase in COND containing P# is "P# IN Q2". To complete this prepositional phrase, we call TRANS(P#, 0, P# IN Q2). This produces "parts", and a call to MOD(P#, 0, P# IN Q2). Looking at how MOD works, we see that this makes a call to MOD(P#, 0, PART.COLOR='RED'). The output from MOD is "colored red".

Back in the table, we go to the next line, containing J#. Now J# occurs in COND, so output "to". Call TRANS(J#, 0, SHIPMENT.J#='J1' OR SHIPMENT.J#='J2'). NOUNS gives us "projects". Call MOD(J#, 0, SHIPMENT.J#='J1' OR SHIPMENT.J#='J2'). Since the major connective is "OR", we call MOD(J#, 0, SHIPMENT.J# ='J1'), output "OR", and call MOD(J#, 0, SHIPMENT.J#='J2'). The first MOD gives "project J1", and the second gives "project J2".

<u>Example 11</u>:

```
SELECT S#
FROM   SHIPMENT
WHERE J# IN
         (SELECT J#
          FROM   PROJECT
          WHERE  PROJECT.JCITY='LONDON'
          OR     PROJECT.JCITY='PARIS')
AND   P# IN
         (SELECT P#
          FROM   PART
          WHERE  PART.COLOR='RED');
```

For this example, the translation will be: "Select the S#-values of suppliers who sent a shipment of parts colored red to projects located in London or located in Paris."

Here A is S#, R is SHIPMENT, and COND is everything after the first "WHERE". Again we use the simpler translation scheme. Step 1 gives us "Select the S#-values of", step 2 gives us "suppliers", and step 3 gives us "who sent a shipment of parts colored red to projects located in London or located in Paris".

CHAPTER VI

APPLICATIONS

We stated earlier that the objective of this thesis is to present a method to interpret an SQL query and translate it into natural language. At first, this may seem a bit odd, in view of the current trend of translating natural language to a database language such as SQL. In this chapter, we outline some potential applications of our research results as described in the previous chapters.

(i) SQL Tutorials: Consider the design of a computer-based tutorial system to teach programmers and users how to use SQL to interact with the database system. Most likely there will be a drill session following a brief introduction to the essential features of the language. The trainee will be given a sample database schema such as our Part-Supplier-Project schema. Then for each request of information, such as "find all suppliers who supply at least one red part", the trainee is asked to formulate an SQL query correctly. The least the tutorial system should do is to determine whether the submitted query is syntactically correct, a task an SQL interpreter/compiler must be able to perform. Thus, we can assume the syntax of the query is correct. A highly desirable feature of such a tutorial system is to provide feedback to the trainee regarding the correctness of the query, i.e., whether the query will do what it is supposed to do. One possible way of providing such a facility is to have the tutorial system use the query to retrieve information from a sample database and allow the trainee to check the answer by examining the entire sample database. One of the pitfalls of this approach is that the answer might be correct even if the query is incorrect, which may happen if the sample database is very small. The alternative is to make use of our proposed system to inform the trainee of the English translation of the query in a clear and unambiguous way. To take this approach one step further, it is possible to expand our system into an expert system which can diagnose a submitted query and suggest ways to modify the query.

(ii) <u>SQL Programming Aid</u>: The same feedback mechanism described above will also be helpful to experienced end-users and programmers who want to exploit the full retrieval power of SQL. This mechanism can be embedded into the database system as an option for debugging purposes. Experience has shown that most obscure program bugs occur because the program neglects to handle "boundary" conditions properly. Example 9 will illustrate our point here. After the interpretation of QT, we obtain the following psuedo-SQL query:

```
SELECT J#
FROM   SHIPMENT SHIPMENTX
WHERE (SELECT P#
       FROM   SHIPMENT
       WHERE  S#='S1'
       AND    J#=SHIPMENTX.J#)
      CONTAINS
      (SELECT P#
       FROM   SHIPMENT
       WHERE  S#='S1')
```

The "boundary" condition of this query is the condition that S1 supplies no parts to any projects. When this condition is true, which means the 'CONTAINS' predicate is always true, all J#'s will be retrieved as the answer. This boundary condition can be detected by our system and displayed to the programmer as part of the feedback. In this case, the sentence "...or S1 supplies no parts to any projects" is appended to the normal output.

(iii) <u>Syntactic</u> <u>Query</u> <u>Optimization</u>: This is in contrast to semantic query optimization [King81], which makes use of knowledge of data semantics to speed up query processing. The latter is very dependent on the application domain because the semantic knowledge is expressed in the form of semantic integrity constraints of the database, such as "Every project located in London must be supplied by local suppliers". Syntax-based query optimization has been suggested by other database researchers. [KimW82], for example, proposed some rules by which certain types of queries can be transformed into other queries and showed the transformation can reduce processing time. The rules we have proposed for the QT can be applied with similar results. For example, the conversion of double 'NOT

EXIST' into 'CONTAINS' reduces the number of levels of the query by one. Still greater savings can be obtained by interpreting queries. Example 7 is a case in point. To retrieve the answer to this query, we need only to look for tuples with the same value of S# but different values of J#. Thus one pass of the relation SHIPMENT will be sufficient. A secondary index on S# or sorting the tuples by S# will help, but neither of them is strictly necessary. To further illustrate this approach, consider the following complicated query:

```
SELECT   S#
FROM     SHIPMENT  SHIPMENTX
WHERE    S# NOT IN
        (SELECT  S#
         FROM    SHIPMENT
         WHERE   J# IN
                (SELECT  J#
                 FROM    SHIPMENT
                 WHERE   P# ¬= SHIPMENTX.P#))
```

After the processing by QT, this query becomes a psuedo-SQL query as follows:

```
SELECT   S#
FROM     SHIPMENT  SHIPMENTX
WHERE    S# IN
        (SELECT  S#
         FROM    SHIPMENT
         WHERE   FOR ALL SHIPMENT: J# IN
                (SELECT  J#
                 FROM    SHIPMENT
                 WHERE   FOR ALL SHIPMENT:P#=SHIPMENTX.P#))
                       (or equivalently, NUMBER(P#)=1)
```

The translation of this query is: "Find suppliers such that each project such a supplier supplies uses only one part". Of course, our proposed system can identify this query by its syntax, without actually deriving its meaning, which is application dependent. In fact, this query belongs to the category of queries without 'EXIST', has interblock reference, and is of the type (-,+,-) in terms of negation by levels. For this class of queries, there will be a predetermined search strategy with the attributes involved (in this case S#, J# and P#) as the input parameters. In order to estimate the processing time of this strategy, we assume the relation SHIPMENT is sorted by S# and then by J#. Since the primary key of the relation consists of S#, J# and P#, any two tuples with the same value of S# and J# must have different values under P#. Thus for a value of S# to be included into the answer, all J#'s associated with this value of S# must be different from each other. Again, one pass of the relation SHIPMENT will yield the answer.

(iv) 'Extended' SQL: We have loosely defined Extended SQL to include all forms of transformation output by the QT and, of course, SQL itself. It is not the intention of this thesis to present a precise definition of Extended SQL, but we believe it could form the basis of an improved language interface. The advantages are obvious. It allows the users to use more English-like queries to retrieve a greater variety of information without having to modify the SQL language processor. While we are aware SQL is not a perfect database language (as [Date84] points out) there ~~are~~ already plenty of existing database systems that are using SQL or SQL-like languages. For these systems, a front-end can be written to accept psuedo-SQL queries and then transform them into a form accepted by the SQL language processor of the database system. In fact, a future research direction is to pursue the idea of extending SQL to bring it is as close to natural language as possible, while keeping its characteristic of application independence intact.

CHAPTER VII

CONCLUSION AND PROSPECTS FOR THE FUTURE

The ELFS system is based on Codd's work with tables, but we have made several extensions. The idea of a Query Transformer is a new contribution. The transformations we discovered are new, and we were able to prove several theorems which rigorously established the correctness of these transformations. The ELFS system goes beyond previous work in its ability to handle universal quantification, the general use of negation, and the use of 'OR' and 'AND'.

Although SQL is rapidly becoming the de facto standard for data language for relational database systems, it has many shortcomings. [Date84] has presented a critique of the language. Our concerns here focus on the untapped power of the database language for casual users. Even for simple, day-to-day and uncontrived requests for information, one has to resort to complex, unnatural, and therefore hard-to-understand queries. These concerns have motivated us to develop a system to translate an SQL query into natural language. The obvious applications of this system will be found when developing computer-based SQL tutorial systems and SQL debugging aids. During the process of translation, knowledge about the syntax of the query is acquired, which will be useful in query optimization. The whole exercise described in this thesis could lead to construction of a more user-friendly, restricted natural language front-end to the existing SQL language, which can easily migrate from one application domain to another.

An interesting topic for further research would be to see how easily one could transport the ELFS system to a different database. Also, tools could be developed to aid in the construction of ELFS systems for different applications. There should be a systematic way of generating the tables used in the translations. There is much more work that could be done on the algorithms for QT and NLG. In addition, ELFS could be generalized to include interblock reference. Finally, one could investigate queries with more than three levels.

References

[Atze82]
Atzeni, P. and D.S. Parker, "Assumptions in Relational Database Theory", Proc. of ACM Symp. on Principles on Database Systems", 1982, pp. 1-9

[Brad82]
Bradley, J., "File and Data Base Techniques", Holt, Rinehart & Winston, New York, 1982

[Codd70]
Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, No. 6, Vol. 13, 1970, pp. 377-387

[Codd78]
Codd, E.F. et. al., "Rendezvous Version 1: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases", Research Report RJ2144, IBM Research Lab., San Jose, Calif., 1978

[Cham76]
Chamberlin, D.D. et. al. "SEQUEL 2: A unified Approach to Data Definition Manipulation, and Control", IBM J. Research & Developments, Nov. 1976, pp.560-575

[Date81]
Date, C.J., "An Introduction to Database Systems", Addison-Welsey, Readings, Mass., 1981

[Date84]
Date, C.J., "A Critique of the SQL Database Language", ACM SIGMOD Record, No. 3, Vol. 14, 1984

[KimW82]
Kim, W. "On Optimizing an SQL-like Nested Query", ACM TODS, No. 3, Vol. 7, 1982, pp.443-469

[King81]
King, J.J., "QUIST: A System for Semantic Query Optimization in Relational Databases", Proc. of VLDB, 1981, pp. 510-517

[Shne78]
Shneiderman, B., "Improving the Human Factors Aspect of Database Interactions", ACM TODS, No. 4, Vol. 3, 1978, pp.417-439

[Ullm82]
Ullman, J.D., "The U.R. Strikes Back", Proc. of ACM Symp. on Principles of Database Systems, 1982, pp. 10-22.

Appendix : Prolog program.

We give a listing of the Prolog program, which includes some comments. In this program, the relations of the sample database were given the shorter names S, P, J, and SPJ. At the end, a sample of the output is given.

We can make some comparisons between the Prolog program and the PL/I program. Many string manipulation routines had to be written in PL/I which are already built-in in Prolog. If we exclude all these utilities from the PL/I program, it is still about four times as long as the Prolog program. The pattern-matching facilities of Prolog eliminate a whole series of IF-THEN-ELSE clauses in the central routines of ELFS. There was some concern that Prolog might not be fast enough, but this proved to be unfounded. For our sample database, the processing time was under 1 second per query.

```
/*****************************************************/
/* This part contains some of the simple functions.  */
/* These are domain dependent.                       */
/*****************************************************/

/* ATOREL finds the relation that an attribute belongs to. */

  ATOREL(S,S).
  ATOREL(SNAME,S).
  ATOREL(STATUS,S).
  ATOREL(SCITY,S).
  ATOREL(P,P).
  ATOREL(PNAME,P).
  ATOREL(COLOR,P).
  ATOREL(WEIGHT,P).
  ATOREL(PCITY,P).
  ATOREL(J,J).
  ATOREL(JCITY,J).
  ATOREL(JNAME,J).

/* RELATION is true if its argument is a relation of the  */
/* database.                                              */
/* ATTRIB is true if its argument is an attribute.        */

  RELATION(S).
  RELATION(P).
  RELATION(J).
  RELATION(SPJ).

  ATTRIB(S).
  ATTRIB(SNAME).
  ATTRIB(STATUS).
  ATTRIB(SCITY).
  ATTRIB(P).
  ATTRIB(PNAME).
  ATTRIB(COLOR).
  ATTRIB(WEIGHT).
  ATTRIB(PCITY).
  ATTRIB(J).
  ATTRIB(JCITY).
  ATTRIB(JNAME).
  ATTRIB(QTY).
```

```
/* RELNOUNS provides the noun corresponding to a relation.  */
/* NOUN provides the nouns for attributes and for the keys  */
/* of relations.                                            */

  RELNOUN(S,suppliers).
  RELNOUN(P,parts).
  RELNOUN(J,projects).

  NOUN(S,S_values).
  NOUN(P,P_values).
  NOUN(J,J_values).
  NOUN(SNAME,names).
  NOUN(STATUS,statuses).
  NOUN(SCITY,cities).
  NOUN(PNAME,names).
  NOUN(COLOR,colors).
  NOUN(WEIGHT,weights).
  NOUN(PCITY,cities).

/* Utilities */

  PRINT(*X.*Y)<- WRITECH(*X) & WRITECH(' ') & PRINT(*Y) &/.
  PRINT(NIL). /* <-WRITECH('nil ').  */

  APPEND(NIL,*L,*L).
  APPEND(*X.*L1,*L2,*X.*L3)<-APPEND(*L1,*L2,*L3).
```

```
/* ALTER performs the basic translation algorithm of NLG.   */
/* It can handle both level 1 and level 2 queries.          */

   ALTER(SELECT.*A.FROM.*RLIST.WHERE.*COND,*OUT)
<- ATTRIB(*A) & NOUN(*A,*NOUN)
 & ATOREL(*A,*RA) & RELNOUN(*RA,*RNOUN) & /
 & NEWLINE
 & PRINT(_.Start.of.new.query._.NIL) & NEWLINE &NEWLINE
 & PRINT(Query.is.SELECT.*A.FROM.*RLIST.WHERE.*COND)
 & MOD(*RA,*COND,*MOD)
 & TABLES(*A,*RLIST,*COND,*TABLES)
 & APPEND(Select.the.*NOUN.of.*RNOUN.NIL, *MOD, *OUT1)
 & APPEND(*OUT1,*TABLES,*OUT)
 & NEWLINE &PRINT(Translation.is.:.NIL) & NEWLINE
 & PRINT(*OUT)
 & NEWLINE & NEWLINE.


/* MOD(*RA,*COND,*MOD)                                        */
/* returns in *MOD a list of all modifiers of RNOUN           */
/* in *COND.                                                  */
/* It calls MODA for each possible attribute of that noun.    */

/* Check for occurence of a "FOR ALL".                        */

MOD(*R,FOR.ALL.X.IN.*RX.*RY.*BY.=.*RZ.*B.IMPLIES.*COND,*OUT)
<- NOUN(*B,*NOUN) & ATOREL(*B,*RB) & RELNOUN(*RB,*RNOUN)
 & MOD(*RB,*RB.*B.=.*NOUN.NIL,*MOD1)
 & MOD(*RB,*COND,*MOD2)
 & APPEND(having.the.prop.that.for.all.*RNOUN.x.if.x.is.NIL,
          *MOD1,*OUT1)
 & APPEND(then.x.is.NIL,*MOD2,*MOD22)
 & APPEND(*OUT1,*MOD22,*OUT).

/* Break up the AND's.                                        */

MOD(*RA,*C1.AND.*C2,*MOD)
<- MOD(*RA,*C1,*M1)
 & MOD(*RA,*C2,*M2)
 & APPEND(*M1,*M2,*MOD).

/* Now *COND should contain only a single condition.          */

MOD(S,*COND,*MOD)
<- MODA(S     ,*COND,*S)
 & MODA(SNAME, *COND,*SN)
 & MODA(STATUS,*COND,*ST)
 & MODA(SCITY, *COND,*SCITY)
 & APPEND(*S,*SN,*L1)
 & APPEND(*L1,*ST,*L2) & APPEND(*L2,*SCITY,*MOD).
```

```
MOD(P,*COND,*MOD)
<- MODA(P      ,*COND,*P)
 & MODA(PNAME, *COND,*PN)
 & MODA(COLOR, *COND,*PC)
 & MODA(WEIGHT,*COND,*WT)
 & MODA(PCITY, *COND,*PCITY)
 & APPEND(*P,*PN,*L1) & APPEND(*L1,*PC,*L2)
 & APPEND(*L2,*WT,*L3)& APPEND(*L3,*PCITY,*MOD).

MOD(J,*COND,*MOD)
<- MODA(J      ,*COND,*J)
 & MODA(JNAME,*COND,*JN)
 & MODA(JCITY,*COND,*JCITY)
 & APPEND(*J,*JN,*L1)
 & APPEND(*L1,*JCITY,*MOD).

/* MODA 'goes into' any IN-clause. */

MODA(*A,*A.IN.SELECT.*B.FROM.*RLIST.WHERE.*COND,*MODA)
<- ATOREL(*A,*RA)
 & MOD(*RA,*COND,*MOD)
 & TABLES(*B,*RLIST,*COND,*TABLES)
 & APPEND(*MOD,*TABLES,*MODA).

MODA(S      ,*.S.=.*S.NIL,              *S.NIL).
MODA(SNAME, *.SNAME.=.*SNAME.NIL,  named.*SNAME.NIL).
MODA(STATUS,*.STATUS.=.*STATUS.NIL,having.a.status.*STATUS.NIL).

MODA(SCITY, *.SCITY.=.*SCITY.NIL,  located.in.*SCITY.NIL).

MODA(P      ,*.P.=.*P.NIL,            *P.NIL).
MODA(COLOR, *.COLOR.=.*C.NIL,     colored.*C.NIL).
MODA(PNAME, *.PNAME.=.*PN.NIL,    named.*PN.NIL ).
MODA(WEIGHT,*.WEIGHT.=.*WT.NIL,   weighting.*WT.NIL).
MODA(PCITY, *.PCITY.=.*PCITY.NIL,made.in.*PCITY.NIL).

MODA(J      , *.J.=.*J.NIL,          *J.NIL).
MODA(JNAME, *.JNAME.=.*JNAME.NIL,named.*JNAME.NIL).
MODA(JCITY, *.JCITY.=.*JCITY.NIL,located.in.*JCITY.NIL).
MODA(*,*,NIL).
```

```
/* Distribute TABLES over the relation list. */

TABLES(*A,*R.*REST,*COND,*T)
<- TABLE(*A,*R,*COND,*T1)
 & TABLES(*A,*REST,*COND,*T2)
 & APPEND(*T1,*T2,*T).

TABLES(*A,NIL,*,NIL).

/* SPJ table (domain dependent)        */
/* *MP contains the modifiers for P. */
/* *NPP is the noun phrase.           */

TABLE(S,SPJ,*COND,*TABLE)
<- QTY(*COND,*QTY)
 & MOD(P,*COND,*MP) & P(S,*MP,*NPP)
 & MOD(J,*COND,*MJ) & J(*MJ,*NPJ)
/*  & PRINT(MODP.=.NIL) &PRINT(*MP) &NEWLINE
   & PRINT(MODJ.=.NIL) &PRINT(*MJ) &NEWLINE
   & PRINT(*NPP) & NEWLINE
   & PRINT(*NPJ) & NEWLINE  */
 & APPEND(who.sent.a.shipment.NIL,*NPP,*OUT1)
 & APPEND(*OUT1,*QTY,*OUT2)
 & APPEND(*OUT2,*NPJ,*TABLE).
/* &PRINT(*OUT1) & NEWLINE
  &PRINT(*OUT2) & NEWLINE. */

TABLE(P,SPJ,*COND,*TABLE)
<- QTY(*COND,*QTY)
 & MOD(S,*COND,*MS) & S(*MS,*NPS)
 & MOD(J,*COND,*MJ) & J(*MJ,*NPJ)
 & APPEND(supplied.NIL,*QTY,*OUT1)
 & APPEND(*OUT1,*NPS,*OUT2) & APPEND(*OUT2,*NPJ,*TABLE).

TABLE(J,SPJ,*COND,*TABLE)
<- QTY(*COND,*QTY)
 & MOD(P,*COND,*MP) & P(J,*MP,*NPP)
 & MOD(S,*COND,*MS) &   S(*MS,*NPS)
 & APPEND(supplied.NIL,*NPP,*OUT1)
 & APPEND(*OUT1,*QTY,*OUT2) &APPEND(*OUT2,*NPS,*TABLE).

/* S,P and J tables are NIL                                  */

TABLE(*,S,*,NIL).
TABLE(*,P,*,NIL).
TABLE(*,J,*,NIL).
```

```
/* The SPJ table calls the functions below to form noun */
/* phrases from the modifiers it has found.             */

  S(NIL,NIL).
  S(*L,*NPS)<-APPEND(by.suppliers.NIL,*L,*NPS).

  P(S,NIL,NIL).
  P(S,*L,*NPP)<-APPEND(of.parts.NIL,*L,*NPP).

  P(J,NIL,NIL).
  P(J,*L,*NPP)<-APPEND(with.parts.NIL,*L,*NPP).

  J(NIL,NIL).
  J(*L,*NPJ)<-APPEND(to.projects.NIL,*L,*NPJ).

  QTY(*U.QTY.*COMP.*CONST,in.a.quantity.*COMP.*CONST).
  QTY(*U,NIL).

/* level 2 example                                        */


 <-ALTER(SELECT.S.FROM.(SPJ.NIL).WHERE.P.IN.
    SELECT.P.FROM.(SPJ.NIL).WHERE.SPJ.S.=.S1.NIL,*OUT).

/* end of the Prolog program. */
```

Here is the output produced by the above program.

```
 #Execution begins   21:37:26
  Prolog/MTS 0.2

  _ Start of new query _

  Query is :
  SELECT S FROM SPJ.NIL WHERE P IN SELECT P FROM SPJ.NIL
  WHERE SPJ S = S1

  Translation is :
  Select the S_values of suppliers who sent a shipment of parts
  supplied by suppliers S1
```