DESIGN OF A SUITABLE TARGET LANGUAGE FOR NATURAL LANGUAGE INTERFACES TO RELATIONAL DATABASES

by

Christiana I. Ezeife

B.Sc University Of Ife, Nigeria 1982

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

MARCH 1988

APPROVAL

Name: Christiana I. Ezeife

Degree: Master Of Science

Title of thesis: Design Of A Suitable Target Language For Natural Language Interfaces To Relational Databases

Examining Committee:

Chairman: Dr. Binay Bhattacharya

Dr. W. S. Luk
Senior Supervisor

Dr. Veronica Dahl

Dr. Nick Cercone
External Examiner
School of Computing Science
Simon Fraser University

Date Approved: March 1988

Title of Thesis/Project/Extended Essay

Design of a Suitable Target Language for Natural Language Interfaces to Relational Databases

Author:

[illegible]
(signature)

CHRISTIANA I. EZEIFE
(name)

April 11, 1988
(date)

ABSTRACT

Natural language interfaces (NLIs) to databases are an important technological advance because they enhance access to databases for non-specialists and casual users such as office workers, managers and consumers. Though some have argued that there are good alternatives to the use of natural language (NL) as an interface to databases, NLIs to databases provide an easy means of communication with databases for all including naive users. However, to increase the commercial acceptance of NLIs to databases, there is a need to increase the capabilities of existing systems. In addition, there is a need to make the target query language easy to read, in order to increase the confidence of users in the correctness of the translations.

One of the most sophisticated NLI systems today is the Transformational Question Answering system (TQA), which is a domain-independent English interface to IBM SQL-based program products. TQA's target language provides some elegant extensions to SQL that enable the system to handle more classes of English language queries than SQL. TQA, however, retains one of SQL's shortfalls, which is that it is not a very easily understandable language. One major reason for this is that it lacks an explicit construct for a universal quantifier and has to rely on double or nested negation to simulate it.

In this thesis, a method is provided for translating NL queries into a target language which is easily convertible to a simplified SQL syntax embedded in a host language program. With this method, we are able to solve more classes of queries than TQA and avoid the use of double or nested negation to simulate universal quantification. Two algorithms are involved in the transformation procedures. In addition to presenting these algorithms, the target language is defined, and the modified Logical Form and Canonical query representation are also presented.

ACKNOWLEDGMENTS

I am extremely grateful for the continuing support and guidance of my committee and in particular my senior supervisor, Dr. W. S. Luk who was always there to discuss my research problems with me. Special thanks also go to Mr. Paul McFetridge and Mr. Gary Hall for the helpful discussions they offered; and for allowing me use the Canonical query representation from the System X they are developing currently at Laboratory For Computer and Communications Research (LCCR) at SFU. My thanks also go to Dr. Nick Cercone who is the principal investigator of the System X project together with Dr. W. S. Luk who is the co-investigator of the same project. I would also like to acknowledge the partial financial support of this research by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the strategic grant for the System X project. My sincere thanks also go to Ms. Sharon Hamilton for helping to edit this thesis. I am also very grateful to my husband, Dr. A. N. Ezeife for his moral support and care, and my daughters Emilia and Doreen for being patient with me during this research period.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 What is a Natural Language Interface to a Database System ?

A natural language interface (NLI) to a computer database provides users with the capability of obtaining information stored in the database by querying the system in a natural language (NL) [1] [MAGP 85]. An NLI to a database system is a system that accepts natural language queries from the user, and translates these queries into some intermediate form, and then to some formal query language before actual data retrieval. Natural Language Interfaces (NLIs) to databases make all these transformations without the user having to know about the particularities of the database structure.

1.2 Motivation

Natural language interfaces are desirable as opposed to formal query languages because:

1. People already know a NL and do not have to learn an artificial language.
2. NL provides logical data independence since the user does not need to know about the structure of the database.
3. In the future, spoken input will probably be preferred to typed input by users, and NL systems will be a necessary component of such a voice system.

Typically, NLIs to relational databases translate NL input to an intermediate form called the *logical form* (LF) which expresses the system's understanding of an input query. The logical form is then translated into a query in the formal query language of the database. One of the fundamental limitations with these formal query languages is that they fail to handle linguistic devices such as

[1] A natural language is a language spoken by a group of people, e.g., English.

anaphora and ellipses [JKVSTW 85] [2]. Thus, these NLIs are restricted by their target query languages. This limitation apart, NLIs to relational databases are important because of the growing importance of the relational model [Codd 70] which is easier to use than the other main models of database design [Ullman 82]. Secondly, many database management systems (e.g., INGRES [SWKH 76], QBE [Zloff 77] and SQL/DS) have used the relational model as a basis.

Structured Query Language (SQL) is becoming the de facto standard query language for relational database systems [3]. Thus, we are motivated to work on an NL interface which translates its input to SQL. SQL however, has some shortcomings like other database languages some of which are discussed in [Date 86b].

Transformational Question Answering system (TQA) [Johnson 84] is an English interface to IBM SQL-based program products [Johnson 84], [Petrick 84] and [Damerau 85]. We have chosen to work with the TQA system in particular because it overcomes many of the restrictions of the formal query language, SQL by providing some elegant extensions to SQL. However, TQA still suffers from an obvious shortfall of SQL, which is that some SQL constructs are hard to understand because universal quantification is simulated with double or nested negation. [LukKl 85] also show that not only are these SQL constructs hard to understand but the constructs generated are not always valid.

System X is a NLI to SQL-based relational databases currently under development at the Laboratory for Computers and Communications Research (LCCR) at Simon Fraser University. System X is similar to TQA in many respects [MHCL 87]. In this research, we shall be using System X as another implementation of the TQA system.

[2] Anaphoric queries are queries that need results of previous queries. Ellipses are fragmentary queries which need to reuse words from previous queries. For example, a preceeding query is "which math majors take math102?". An anaphoric query posed after this query is "who got the highest mark ?", and an ellipse is "computing majors?" meaning "which computing majors take math102 ?".

[3] A brief description of a relational database system is given in Section 3.1

Due to the inherent limitations of SQL, most commercial enterprises use *embedded SQL* when communicating with their databases. Embedded SQL consists of simple SQL statements embedded in programs written in some high level language. [Stone 88] observed that human factors studies and early usage of relational systems has shown clearly that end users prefer customized interfaces written by programmers to SQL. This is because the customized interfaces are made appropriate to the user's application needs. Thus, the interest of these commercial enterprises in adapting NLIs for their existing databases will be enhanced if NLIs can translate their NL inputs to embedded SQL rather than SQL.

Two things are desirable if the commercial acceptibility of NL query systems is to be increased.

1. The systems have to be made as expressive as possible so as to provide a wider linguistic coverage. This will allow the naive users in the organizations to carry out at least the same type of tasks that they were able to do with formal query language systems (e.g., SQL embedded in a host language program) if the organizations switch to the use of NLI systems.
2. The database administrator (DBA) and the application programmers in the organizations must have a high degree of confidence in the transformational procedures and the correctness of the results from the NLI system. This will assure them that switching to NLIs will still provide the desired and correct response to the naive user.

To achieve the above two objectives, it is essential to develop methods to make the NLI systems more expressive and their intermediate results easy to understand.

1.3 The Task

The objective of this thesis is to take a step towards increasing the commercial acceptibility of NLIs. We argue that it will be more beneficial to translate NL queries to embedded SQL using only easy-to-understand SQL constructs so that users[4] will have confidence in the transformations.

Achieving this objective involves formalizing a target query language into which NL queries can be mapped. Our target language is designed such that it is easily convertible to embedded SQL. With System X, an English language query is parsed and semantically analysed to generate an intermediate form called *Canonical Query Representation* (CQR). A CQR is a tree that explicitly defines the scoping of quantifiers in the query, and also shows the join paths among relations in the database schema. Although we have defined some extensions to the current capabilities of the System X CQR, it is outside the scope of this research to explain the procedures involved in transforming the English language query to the extended CQR. TQA and System X first transform CQR to a second intermediate form, LF, and then translate LF to a target language consisting of the full SQL syntax plus a few additional constructs. Our approach is quite different from TQA approach. Instead of adopting the full SQL syntax plus a few more constructs, we provide control constructs commonly found in high-level programming languages (e.g., FOR-LOOP and IF-THEN-ELSE), coupled with a much simplified SQL syntax. The advantages of our approach include:

1. With the control constructs explicitly defined, instead of buried deep down in the SQL itself, the language can be more easily extended to provide more features than SQL has to offer. For example, many features of SQL are designed to provide easy reporting; GROUP-BY and some aggregate functions are examples. However, there are severe limitations in the report formatting.
2. The whole system will be more portable since our target language is more adaptable to different implementations of relational database systems, and has a much reduced dependency on the idiosyncrasies of SQL.
3. The target language is designed such that output of the translation of the input, after

[4] By users here, we mean either the database administrator or the application programmer and not necessarily an end user.

syntactic and semantic analysis resembles a program written in the syntax of the target language, which is a superset of TQA's "ad hoc" target language.

The major tasks involved in the research are:

* Providing some extensions to the structure of the CQR used by System X to handle more classes of English language queries than System X handles currently.
* Modifying the structure of the LF used by TQA and System X, to accomodate our extensions.
* Presenting the BNF description of the target language.
* Defining an extensive algorithm which accepts CQR as input and generates LF.
* Defining a second algorithm which maps our LF to our target language.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the literature review of some existing NL systems, discussing their capabilities and what needs to be done in order to increase the commercial acceptance of these systems. Chapter 3 discusses some basic database terms, embedded SQL and its use, as well as TQA's target language, while Chapter 4 discusses System X and our extensions to some modules of that system. Chapter 5 discusses features of our target language and the simplified version of SQL that we use. Chapter 6 discusses the transformational algorithms in our system while Chapter 7 presents conclusions.

CHAPTER 2

NATURAL LANGUAGE INTERFACE TO DATABASE SYSTEMS

In this Chapter, we present a general description of some existing NLIs to databases. In particular, we discuss their features, and describe their advantages and disadvantages over formal query languages. The capabilities of these NLIs are also discussed.

2.1 Advantages and Disadvantages of NLIs

[PyKi 85] observed that a good goal in the commercial use of artificial intelligence (AI) is to make the machine know more about the user so that the user will need to know less about the machine. This slogan highlights the point that if the user is reluctant to learn the formal languages associated with the use of computers, he needs the computer to know about him and to provide him with tools so that he can use it. Since the user already knows how to communicate in a NL, NLIs provide a good tool for communication between the naive user and the computer.

Although current NLIs seem to represent a great technological advance, these systems are still faced with problems, including:

1. The linguistic coverage of the systems is not yet very wide. Some of the systems are not yet able to handle natural language queries involving pronoun references, more than two quantifiers, anaphoric queries and ellipses.
2. Some of the NLIs to relational database systems are restricted by the capabilities of the formal query language to which they map their NL queries.
3. The problems of interpreting queries in most NLIs, have only been solved in an ad hoc way for narrow [5] relational databases, and the customization of such natural language systems

[5] This means for specific relational database schemas without being readily generalisable to other relational database schemas.

to new databases or subject areas represents a serious investment of time and effort [SlJu 85].

2.2 <u>Features of Some NLI Systems</u>

We have chosen some of the existing [PyKi 85] NLI systems to briefly discuss their features and capabilities. The sample systems are:

* the User Specialty Language (USL) [LOZ 85],
* the Transportable English Database Access Medium (TEAM) [MAGP 85],
* a Simple Knowledgeable System (ASK) [BoFr 85],
* the Layered Domain Class System (LDC) [BLT 84], and
* the Transformational Question Answering System (TQA) [Damerau 85].

2.2.1 Capabilities of Current Natural Language Systems

Current NL systems have the capability of answering complete self-contained grammatical questions. Some of the systems can also understand user inputs containing simple pronoun references or minor grammatical errors, certain cases of ellipses, and certain definitions introduced by the user in interaction with the NL system. However, they incorporate only a very limited theory of the application domain, do not translate the query into a general logical form from which inferences can be carried out, and in general are not capable of analysis at the level of discourse pragmatics, which requires that the system maintain a model of the user's needs and intentions. That is, the LF is not general enough to represent the user's presupposition.

<u>1.</u> <u>System Structure</u>

NLI systems have the same basic type of structure, which can be viewed as consisting of two major modules:

a. the acquisition module for customization to new applications, which is not discussed further in this thesis, and

b. the NL module, which translates NL queries to some intermediate forms and then to the target query language.

The main differences in the structure of NLIs lie in whether they map to existing database structures, and whether they generate some kind of intermediate structure. The NL module usually consists of a *lexicon*, a *parser*, a *semantic interpreter*, a *translator*, a *retrieval module* and a number of *grammars*. Some systems have additional features. NLIs require world knowledge in order to understand input questions. NLIs have semantic information stored in a lexicon. Since we are using System X as a model of the TQA system, we proceed to give an idea of what the system structure of a typical NLI looks like by giving a brief description of the System X structure. Then, we outline the differences between the structure of these existing NLI systems outlined above.

System X currently consists of a set of modules which create a CQR from an input query. The modules are a lexicon, a parser and a semantic interpreter. System X's lexicon consists of a syntactic and a semantic dictionary. The semantic dictionary has two parts: a domain-independent dictionary of predicates, operations, quantifiers, etc. which are transported from application to application. In this domain independent dictionary, the meaning of such words as "all, only, not, greater than, and so on" are defined. The second part of the semantic dictionary is an application-dependent dictionary which is largely generated from the database schema. In the application-dependent dictionary, exceptions to the general rules that are applicable in the domain are given. Actual data in a database schema are asserted in the semantic dictionary. System X's parser is a top-down breadth-first parser [MHCL 87]. When an English query is received by the system, the lexicon first replaces every word in the query with its grammatical definition. Next, the parser applies the grammar rules to the query to return all possible parse trees. An example of a grammar rule used is: "a sentence could be a noun, followed by an auxilliary verb, followed by a verb phrase. These parse trees are then sent to the semantic interpreter. The semantic interpreter takes a parse tree, looks at each node, and recursively descends the tree if it is a non-terminal node. If it is a terminal node, it replaces the syntactic definition of that node with its semantic definitions. As it goes up the tree it applies the semantic rules associated with the syntactic rule that created that node. Nouns and adjectives are defined as values in the database; verbs, could be associated with relations. The semantic rules are organized so that those which apply first to the parse tree insert, delete or rearrange nodes in the parse tree to create a CQR. This CQR represents the query in terms of database entities and is first transformed to LF which is intermediate between CQR and SQL. In general, only one parse tree will be semantically allowed as a parse tree is rejected if the semantic interpreter can not allow a meaning representation to it. We now return to the discussion of system structure of existing NLIs.

USL has, in addition to the typical structure, a high level optimizer for SQL queries which removes irrelevant joins and thus improves the response time. TEAM has a scope determiner which allows it to handle a wide range of linguistic structures involving nested quantifiers. TQA has a SQL-to-English language translator which echoes the English language form of the SQL query formulated from the user's query, so the user can verify the correctness of the translation. This feature helps to increase the acceptability of the system. ASK can be used both as a stand-alone system with its own semantic network, and a NLI to an existing database system. USL, TEAM and TQA translate their NL inputs into an internal representation, which is then translated to some database query language. LDC accepts as its input a text file containing database queries in a format less restrictive than the format of relational database systems. USL, TEAM, and TQA all map their NL inputs to a relational query language, because the databases they support are relational. Both USL and TQA translate queries to SQL, while TEAM translates queries to SODA, another relational database query language. LDC, on the other hand, uses a formal query language specifically developed for the text file inputs. This formal query language has not been proved relationally complete. A diagrammatic representation showing a general overview of System X is given in Fig 2-1.

2. Intermediate Structure

TEAM and TQA translate their NL queries to an intermediate logical form which is derivable from first-order logic but extended with certain intensional and high-order operators, and augmented with special quantifiers for definite and interrogative determiners. USL has another kind of intermediate structure which is quite different from logical form. LDC and ASK have no intermediate structures.

3. Linguistic Coverage

ASK is the most advanced NLI that we consider with regard to the classes of the English language that it accepts. It is able to accept anaphoric queries, fragmentary queries, accept queries with pronouns and even give diagnostic messages when it receives ambiguous queries as input. It also allows the user to update the knowledge base, and corrects spelling errors in user input. Among the three systems that map to relational databases, USL, TEAM, and TQA, USL covers the widest range of linguistic structures. USL handles pronouns, quantification, and negation, while TEAM and TQA are poor at handling pronouns and some kinds of quantification. TEAM, however, can produce facts not physically stored in the database, but rather inferred from data in the database. This is an advanced feature which is also found in expert systems.

The linguistic coverage of LDC is the most restrictive of the five systems discussed. All NL inputs to LDC have to be transformed to their noun-phrase equivalents by the user before the queries are posed. For instance, a query like "Which math majors take math100" has to be posed as "math majors taking math100". This is restrictive because some English

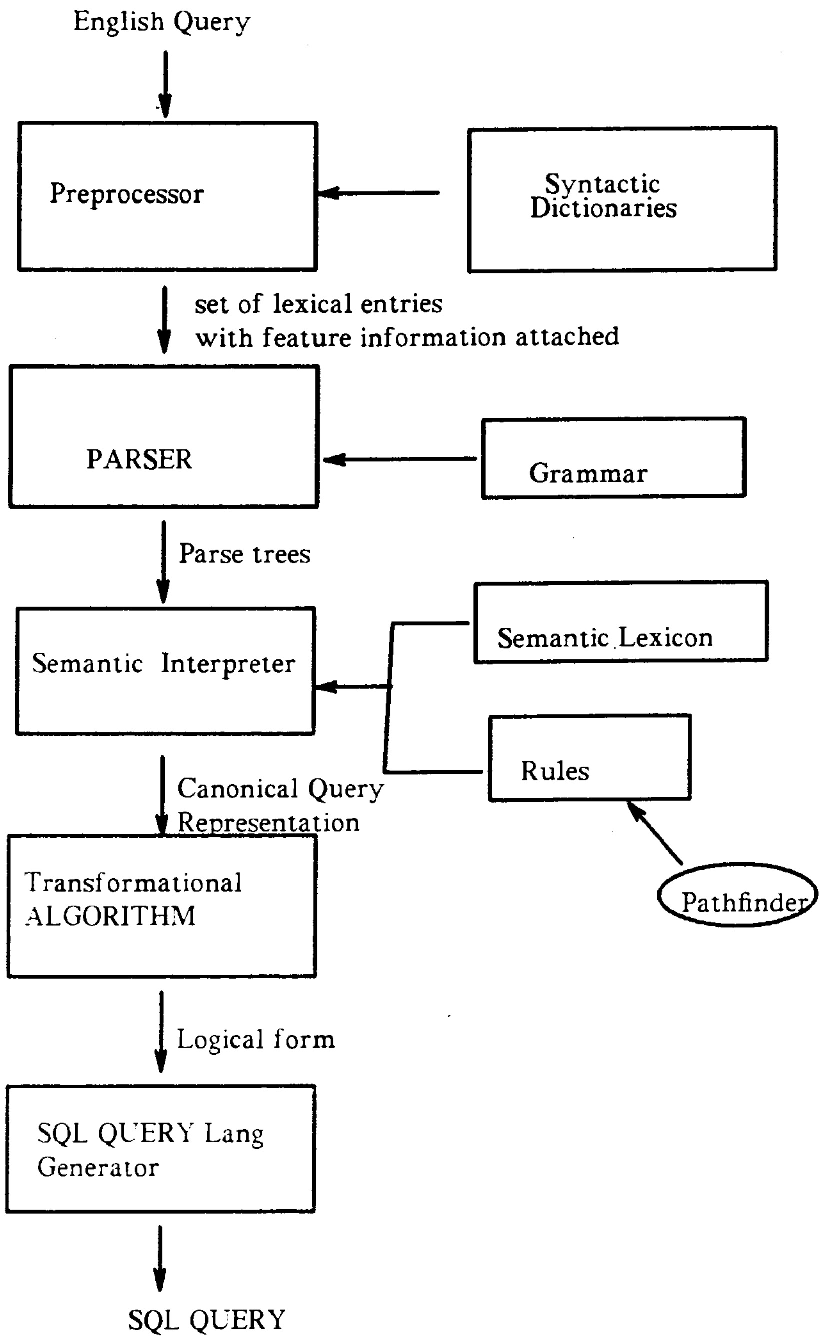


Fig 2-1: Graphical Representation of the System X

language queries might not readily be transformed to a noun phrase, and the user might have trouble doing such transformations. LDC does not allow interrogative queries that require yes/no answers, and cannot handle anaphoric and fragmentary queries or pronouns and negations. LDC does however, have some inferential capabilities: like TEAM, it can retrieve facts not physically stored in the database.

From the linguistic coverage of TEAM as presented in [MAGP 85], it appears that TQA covers a wider range than TEAM.

TQA accepts interrogatives, comparisons, quantifiers, and generates statistical outputs such as histograms, pie charts and bar charts. None of the other four systems including ASK has this form of output.

2.3 Related Work

Although there are many NLIs to databases in existence today, none of the systems that interface to SQL-based relational databases handle universally quantified queries in a systematic easy-to-understand way when translated to SQL. As well, none of the systems translate their natural language inputs into embedded SQL in a host language program.

For cases of quantification, negation and coordination, USL may generate more than one SQL query from a given intermediate structure. The logical form our system uses is a modified version devised to handle more classes of queries and to explicitly interpret quantifiers and their scopes.

In [LOZ 85] and [Damerau 85], it is reported that both USL and TQA translate their English language queries to straight SQL queries, which represent universal quantifiers with double and nested negations. Since these systems are NLIs to existing database systems, we think that in order to overcome their limitations, in particular to cover more sentences and remove the incomprehensibility associated with the SQL query language, it is necessary to translate English language queries to a program-like structure with some simple SQL queries in them. Our system translates the CQR of English language queries to our modified LF, and then to our target language which is easily convertible to simplified SQL embedded in a host language program.

None of the two systems that are NLIs to SQL-based database systems (USL and TQA) can handle conditional queries; but our target language has been designed to handle this important class of queries, and has features to allow processing of such queries as anaphoric queries. We believe that because of our program-like SQL structures intermediate results could be stored as well as previous results.

CHAPTER 3

EMBEDDED SQL AND TQA TARGET LANGUAGES

The target language into which we map our NL queries is designed in such a way that it can easily be converted to embedded SQL. This chapter introduces the reader to some basic database terms before it presents its discussion of SQL and embedded SQL. TQA's target language (TQA-TL), which is an extension of SQL [Date 86a], is also discussed.

3.1 Basic Database Terms

A database is a model of some part of the world; that is, a system which maintains information about an individual or an organization, and makes that information available on demand [Date 86a]. A relational database system is a database system that is perceived by its users as a collection of tables. Fig 3-1 shows an example of a relational database schema, a suppliers-and-parts database which stores information about suppliers and the nature and quantities of parts that they supply. The database consists of three tables, S, P, and SP.

* Table S represents suppliers, each with a unique supplier number (S#) and a supplier name (SNAME), a rating or a status value (STATUS), and a location (CITY).
* Table P represents parts, each with a unique part number (P#) and a part name (PNAME), a color (COLOR), a weight (WEIGHT), and a location where parts of that type are stored (CITY).
* Table SP represents shipments and serves in a sense to connect the other two tables together. Each shipment has a supplier number (S#), a part number (P#), and a quantity (QTY).

Suppliers and parts may be regarded as *entities*, and a shipment may be regarded as *a relationship*

S	S#	SNAME	STATUS	CITY
	S1	Smith	20	London
	S2	Jones	10	Paris
	S3	Blake	30	Paris

P	P#	PNAME	COLOR	WEIGHT	CITY
	P1	Nut	Red	12	London
	P2	Bolt	Green	17	Paris
	P3	Screw	Blue	17	Rome

SP	S#	P#	QTY
	S1	P1	300
	S1	P2	200
	S1	P3	400
	S2	P1	300
	S2	P2	400
	S3	P2	200

Fig 3-1: The Supplier-and-Parts database

between entities, suppliers and parts. Every table, or *relation*,[6] has a unique identifier, called a *candidate key*, for every record in that table. A *tuple* is a row in a relation. The column names of a relation are called *attributes*: For example, S#, SNAME, STATUS and CITY are attributes of the relation S.

3.1.1 The SQL Language

Structured Query Language (SQL) is a language for relational databases. SQL can be used for *data definition* and *data control* (both not discussed further in this thesis) as well as for *data manipulation*. One of the things a user may want to do when manipulating data is data retrieval. SQL can either be used as an interactive query language or embedded in a host language program [7]. The general structure of SQL constructs for data retrieval is given below:

```
SELECT [UNIQUE] <list of attributes>
```

[6] The words table and relation are used interchangeably.

[7] In this thesis, when SQL is not qualified or strict SQL is used, we mean interactive SQL.

```
FROM      <list of relations>
WHERE      (condition(s) is/are true)
```

For instance, the query "what is the city of supplier S2?" is expressed as:

```
SELECT CITY
FROM S
WHERE S# = 'S2';
```

The result of this query is a table which looks like:

```
----
CITY
----
Paris
```

The query "what are the parts supplied by each supplier?" is expressed as:

```
SELECT S.S#, SP.P#
FROM S, SP
WHERE S.S# = SP.S#
ORDER BY S.S#
```

The result of this query is the following table:

```
**   **
S#   P#
**   **
S1   P1
S1   P2
S1   P3
S2   P1
S2   P2
S3   P2
```

In the second example, the relations S and SP are said to be *joined* on the attribute S#. A *virtual relation* is a relation that does not physically exist but is derived from various other physical relations when referenced. A *view* enables the user to define a virtual relation that represents a picture of the database. Any valid SQL statement can be used to create a view by preceeding the SQL construct with the statement:

CREATE VIEW (View name) AS

3.2 Embedded SQL

3.2.1 What is Embedded SQL ?

SQL is both an interactive query language and a database programming language. Any SQL statement that can be used interactively at the terminal can also be used in an application program [Date 86a]. When SQL is used in an application program (like Pl/1 or Pascal), it is called embedded SQL. When an interactive SQL query is executed in a host language program, an INTO clause is used in the query to tell the program where to place the result. The same query of Section 3.1, "what is the city of supplier S2?", when embedded in a host language program becomes:

```
EXEC SQL SELECT CITY
         INTO :XCIT
         FROM S
         WHERE S# = 'S2';
```

where XCIT is a program variable to hold the result of the query. In this thesis, we call a program with some embedded SQL statements a *database program* or an *embedded SQL program.*

3.2.2 Advantages of Making Embedded SQL a Component of NLIs

There are two main reasons why making embedded SQL a component of NLI systems would increase the commercial acceptance of NLIs. These reasons are discussed in turn.

The first reason is that SQL queries sometimes are so difficult to understand that the DBA or application programmer may not be sure of the correctness of the queries. If NLI systems are to acquire wide commercial recognition, it is necessary to make the formal queries generated from the user's [8] NLI input as easy to understand as possible. This becomes even more important in the TQA system that translates SQL queries back to English language for verification by the user, because the hard-to-understand SQL query could lead to hard-to-understand translations by the module. When NL queries are translated into simplified SQL constructs embedded in a host

[8] The end user need not understand or even see the internal translations but the database adminstrator and the application programmer may need to.

language program, they are easier for users to comprehend. We demonstrate this point by discussing the obscure nature of some SQL constructs.

One major cause of incomprehensibility in SQL queries is the use of double or nested negation to represent universal quantification. Consider the following query, adapted from a TQA file[9] :

How many suppliers in Paris supply all red screws ?

<u>SQL QUERY</u>

```
SELECT COUNT(DISTINCT B.SNO)
FROM TQASQL.ZSP B, TQASQL.ZS A
WHERE A.SNO=B.SNO
AND A.CITY='PARIS'
AND NOT EXISTS
   (SELECT DISTINCT D.PNO
   FROM TQASQL.ZP C,TQA.ZSP D
   WHERE C.PNO=D.PNO
   AND C.PNAME='SCREW'
   AND C.COLOR='RED'
   AND NOT EXISTS
      (SELECT DISTINCT E.SNO
      FROM TQASQL.ZSP E
      WHERE D.PNO=E.PNO
      AND E.SNO=B.SNO));
```

This query says : "Count the suppliers who are located in Paris such that there is no red screw that is not supplied by these suppliers". The meaning of the English paraphrase is not obvious, raising uncertainty about its correctness.

By explicitly defining control constructs, instead of burying them deep down in SQL itself, the language will be easier to understand.[10] One may argue that if SQL is hard to understand when representing universal quantifiers, other languages like English and Pascal are also hard to understand sometimes. The issue here is that these other languages like English, have alternative

[9] The relations TQASQL.ZSP B, TQASQL.ZS A and TQASQL.ZP in this query are the same as the relations SP, S and P in Fig 3-1 above.

[10] Examples to demonstrate this alternative approach is in Section 5.3.

representations for the hard-to-understand constructs. For instance, in English, instead of asking the obscure question "count the suppliers who are located in Paris such that there is no red screw that is not supplied by these suppliers", English allows the user to express the same query in an easier-to-understand form "how many suppliers in Paris supply all red screws?". Similarly, in Pascal several nesting of loops might make the program hard to understand but Pascal language allows its user alternative methods (e.g, using procedures to express the same thing). The problem with SQL in this regard is that if its user does not want to use double negation to express universal quantification, there is no alternative representation.

The second reason for preferring embedded SQL programs for NLIs is that SQL is quite restrictive and cannot handle all necessary NL queries.

Many English language queries can not be translated into a single SQL query and thus can not be solved with interactive SQL. In addition, a user may want his answer presented in a format different from that allowed by SQL, e.g., pie charts or bar charts. TQA circumvents this restriction of SQL in a limited fashion by extending SQL.

To show the limitations of SQL, we give some classes of English language queries that SQL is not able to represent.

1. Quota Queries that retrieve the identities of a specific number of objects that satisfy a stated condition. For instance, "Find exactly 2 suppliers each of whom supplies part P2". The closest SQL equivalent to the above query is :

```
SELECT  S.S#, S.SNAME
FROM    P, S, SP
WHERE   SP.P#='P2'
AND     SP.S#=S.S# ;
```

   This retrieves all S#'s of suppliers that supply P2 and not just 2 of them as required.

2. Queries that return a specific number (greater than one) of objects within a set of objects that satisfy a maximum or minimum condition. For instance, "Get the three heaviest parts". The

closest SQL equivalent to this query is:

```
SELECT P.P#, MAX (P.WEIGHT)
FROM  P;
```

However, this retrieves just the part with the highest weight and not the three heaviest parts as desired.

3. Queries that rely on strict conditions and demand alternate answers. For instance, "If S2's city is London, give me his name else give me his name, status and city". This category of English queries can not be represented with SQL because of the "else" part of the conditional. For instance, "If S2's city is London, give me his name", will not pose any problem because it can be paraphrased as : "Give me the name of supplier S2 whose city is London"; which can be expressed in SQL as:

```
SELECT S.S#, S.SNAME
FROM  S
WHERE  S.CITY='LONDON'
AND   S.S#='S2' ;
```

SQL handles "if-then" statements as in the above example because most databases are based on the Closed World Assumption (CWA)[11]. With the CWA, if the city of supplier S2 in the above query is London but unknown to the database, the answer returned is Null. However, we are not concerned with this type of Null ambiguity, which is blocked off using CWA. The point we are stressing here is that although "if-then" statements can be represented with SQL, "if-then-else" statements can not. Users may desire to have the power to specify alternate response depending on the result of the search condition. "IF-THEN-ELSE" query will provide this alternate response. A query like: "IF S2's city is London, give me his name else give me his name, status and city", can not be expressed at all because of the else part of the query.

[11]The CWA assumes that everything in the world being modelled is present as data, so that if something is not present it is assumed not to exist.

4. Queries that need Yes or No answers, such as "Is supplier S2 in Paris?". Queries in this category can also not be represented with SQL because strict SQL does not have facilities for writing out messages. It can just retrieve the desired answer if it is in the stored data or return Null if the desired answer is not in the stored data.
5. Queries that ask for graphic and statistical charting. For instance, "Make a bar chart of the quantities supplied by each supplier". Queries in this category can also not be solved with SQL.

Thus, SQL is not powerful enough to handle all queries that an everyday user of a database may wish to have answered. We argue that the capability to embed SQL in a host programming language like PL/1 or Pascal is required if NLI are to handle these types of queries.

3.2.3 A Sample Solution with Embedded SQL

To enable us present a sample program solution to a natural language query using embedded SQL, we briefly discuss some of the syntactic features of embedded SQL program as discussed in [Date 86a].

1. Embedded SQL statements are prefixed by EXEC SQL, so that they can easily be distinguished from statements of the host language.
2. Host variables are variables of the host language (in our case Pascal) declared in the program. When referenced by SQL statements, the host variable names are prefixed with a colon to distinguish them from SQL field names. These host variables can appear in embedded SQL wherever a constant can appear in interactive SQL. They can also appear in an INTO clause of an SQL statement to designate an input area for SELECT or FETCH. Host variables must have a data type compatible with the SQL data type of the fields they are to be compared with or assigned to or from. Host variables and database fields can have the same name.
3. Feedback information about the succesful execution of any SQL statement is available in an

area called the *SQL communication area* (SQLCA). A field of the SQLCA called SQLCODE holds a numeric status indicator. A SQLCODE value of zero means that the SQL statement executed succesfully; a positive value means that the statement executed but constitutes a warning that no data was found to satisfy the request; and a negative value means that an error occurred and the statement did not complete succesfully. The SQLCA is included in the program by means of an EXEC SQL INCLUDE SQLCA statement.

4. A SELECT statement in SQL causes a table to be retreived that, in general, contains multiple records. To be able to handle a record at a time, embedded SQL uses a cursor. A cursor is a pointer that can be used to run through a set of records.

Example Query

To demonstrate the use and simplicity of SQL constructs embedded in a host language program, let us express the same query we represented with SQL in Section 3.3 above. The query is: "How many suppliers in Paris supply all red screws ?".

A pseudocode algorithm for solving this query is:

```
Step 1: Get the number of different red screws, R.
Step 2: For i = 1 to number of suppliers in Paris,
do;
        Step 2.1: Get the count of all red screws, Pi, supplied by supplier i in Paris.
        Step 2.2: If R = Pi
        then supplier-count = supplier-count + 1
Step 3: Print supplier-count
```

The program for this algorithm is given below. The host language here is Pascal.

```
program SQLembed;
 label 15;
```

```
var

 R       :integer ;
 Pi       :integer ;
 count     :integer ;
 supplieri    :array [1..5] of char;
 moresuppliers :boolean;

begin

 EXEC SQL DECLARE S TABLE
      (S# CHAR(5) NOT NULL,
      SNAME CHAR(20),
      STATUS SMALLINT,
      CITY  CHAR(15));

 EXEC SQL DECLARE P TABLE
      (P# CHAR(5) NOT NULL,
      PNAME CHAR(20),
      COLOR CHAR(20),
      WEIGHT SMALLINT,
      CITY CHAR(15));

 EXEC SQL DECLARE SP TABLE
      (S# CHAR(5) NOT NULL,
      P# CHAR(5) NOT NULL,
      QTY SMALLINT);

 EXEC SQL INCLUDE SQLCA;

 EXEC DECLARE Z CURSOR FOR
      SELECT COUNT(UNIQUE SP.P#),SP.S#
      FROM  P,S,SP
      WHERE  SP.S# = S.S#
      AND   S.CITY= 'PARIS'
      AND   SP.P# = P.P#
      AND   P.COLOR='RED'
      AND   P.PNAME='SCREW'
      GROUP  BY S.S# ;

 {The above declares a cursor to point
 at the set of red screws by each
 Paris supplier              }

EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

 {These three statements will make the
 Precompiler not to insert an "IF condition
 GO TO label" statement after each
 executable SQL statement it encounters.
 NOT FOUND means SQLCODE=100, SQLWARNING
means SQLCODE>0 and SQLCODE ~=100,
SQLERROR means SQLCODE<0        }


EXEC SQL SELECT COUNT (UNIQUE P.P#)
    INTO : R
    FROM  P
    WHERE  P.COLOR='RED'
    AND   P.PNAME='SCREW' ;
{The above keeps the total number of different
 types of red screws that are supplied}

IF SQLCODE 0
 then
  begin
   writeln (SQLCA);
   goto 15;
  end;

EXEC SQL OPEN Z;
   If SQLCODE 0
    then
     begin
      writeln (SQLCA);
      goto 15;
     end;
moresuppliers:=true;
while moresuppliers do
   EXEC SQL FETCH Z INTO :Pi,supplieri;

   case SQLCODE of
     100: moresuppliers := false;
    -100: begin
          writeln (SQLCA);
          goto 15;
         end;
       0: begin
          if R=Pi
           then count:=count + 1;
          end;
  end; {for while}
writeln ('Number of suppliers in
 Paris who supply all red screws
 is', count);

EXEC SQL CLOSE Z;
15: end;
```

3.3 TQA's Extension of SQL

TQA provides an elegant extension to the SQL query language which can express queries in categories (1), (2), (4), and (5) of Sub Section 3.2.2 above. For the purposes of this thesis, we name this extension TQA-TL. However, this extension to SQL can not express queries in category (3) above. That is, this extension does not handle conditional queries that are of "if-then-else" structure. In addition, the TQA target language, like SQL, tends to be incomprehensibile when queries involving universal quantifiers are expressed. We discuss the structure of TQA-TL .

a. Queries that need Yes or No answers.

(i) The first structure of TQA-TL for this class of queries (category (4) of Sub Section 3.2.2) in the TQA-TL is given below:

```
POS

(SQL STATEMENT)

***************
```

For instance, "Is S2 in Paris?".

TQA-TL for this query is given below:

```
POS
SELECT S#, SNAME, CITY
FROM S
WHERE S#='S2'
AND CITY='PARIS';
```

(ii) Another structure for Yes/No queries in the TQA-TL is counter queries, which are identical to POS queries except that they return Yes if the attached SQL query retrieves null, and No otherwise. The structure of this form of extension is as follows:

```
CTR

SQL STATEMENT

*************
```

For instance, the query "Does each supplier supply at least 25 parts?" can be represented with TQA-TL as:

```
CTR
SELECT S#
FROM SP
GROUP BY S#
HAVING COUNT(DISTINCT P#)<25 ;
```

This query says: "If there is any S# that supplies less than 25 parts, then say No, otherwise say Yes".

b. The second form of TQA-TL handles English queries in categories (1) and (2) of Sub Section 3.2.2 which are quota queries asking for the retrieval of a specific number of objects. The structure of the extension is as follows, where the symbol | means or.

```
OPERATOR

(OPERATOR TYPE)

(ARITHMETIC EXPRESSION | VALUE | SQL STMT)

(SQL STMT)

***********************************
```

Operator type can be any of the following:

EQUAL, LESSTHAN, MORETHAN, LESSTHANEQ, MORETHANEQ, PERCENT, CHOOSE N, [CHOOSE 1 TO N .... ORDER BY X DESC], and so on.

For instance, "Are there 3 P2's ?".

```
OPERATOR
EQUAL
3
SELECT SUM(SP.QTY)
FROM SP, P
WHERE SP.P#=P.P#
AND P.P#='P2';
```

For instance, "Are there at least 10 red screws ?".

```
OPERATOR
LESSTHANEQ
10
SELECT COUNT(DISTINCT P.P#)
FROM P
WHERE P.COLOUR='RED'
AND P.NAME='SCREW' ;
```

c. Making Charts: The third extension of TQA-TL handles queries in category (5), which ask for graphics printing. The structure of this form of extension can be either of the following:

(i) BAR CHART WITH 2 AXES SPECIFIED

SQL STMT USING GROUP BY

(ii) PIE CHART WITH 2 AXES SPECIFIED

SQL STMT USING GROUP BY

****

Example 1, "Draw me a bar chart of the number of parts supplied by each supplier".

```
BAR CHART WITH 2 AXES SPECIFIED
SELECT SP.S#, SUM(SP.QTY)
FROM SP
GROUP BY S# ;
```

Example 2, "Draw me a pie chart of the number of suppliers by city".

PIE CHART WITH 2 AXES SPECIFED

```
SELECT CITY,COUNT(DISTINCT S#)
FROM S
GROUP BY CITY;
```

CHAPTER 4

EXTENSIONS TO SYSTEM X

In this Chapter, we present a general description of the structure of the CQR of System X before we discuss our extensions to this CQR. We also discuss briefly the System X LF before we describe our modified LF. Finally, our sample database schemas are presented.

4.1 The System X Canonical Query Representation

A System X CQR is produced by the application of a set of inverse lexical transformations and a set of syntactic transformations to a parse tree of the surface structure of a sentence. A CQR is a tree that explicitly defines the quantifiers in an English language query and their scoping, and also shows the relationship between the English language query and the database stored data. A CQR relates the English language query to the stored data by specifying join paths among relations in the database structure. In CQR, an English sentence is represented by the highest node S ^. This S ^ node has descendant nodes which are from left to right: TYPE, QUANTIFIER, and S or NP. The TYPE node specifies whether the English query asks for the retrieval of data or for a "yes or no" answer to a question. The QUANTIFIER node may have more than one quantifier specified, depending on the number of quantifiers present in the English input. There are three types of quantifiers, sometimes referred to as *quants* each with a variable argument. The quants are either EXIST, WHICH or ALL. For instance, the query "Which math majors take all math courses", will have a CQR with a QUANTIFIER node that looks like the following: WHICH X1, ALL X2. Here, X1 refers to math majors while X2 refers to math courses. The S or NP node of the S ^ node specifies the conditions that apply to these variable arguments of the quants, the join paths in the relations and so on. Each S node has a descendant node which is V, and one or more NP nodes. A V node can have as its index one of the following: a name of a relation in the database, an operator (e.g., MAX, MIN, TOTAL), a predicate (e.g., LESSTHAN),a NOT, an OR, or an AND. An NP node can have an index which is a variable name or a constant, or it may have an S descendant. An NP may also have an S ^ descendant to represent relative clauses. The BNF description of this CQR used by System X is given in Appendix A and CQRs for our example queries are given in Appendix E.

The CQR which our first algorithm accepts as its input conforms to the CQR generated by System X. We have, however provided some extensions to the System X CQR, discussed in the following Section. These extensions are essential to enable us to handle the various kinds of queries already solved by the TQA system and as well as additional ones.

4.2 Extensions to System X CQR

Recall that the objective of this research is to extend CQR and to show how it may be translated into a more powerful target query language than SQL. It is not the responsibility of this research to explain the procedures involved in transforming the English language query to extended CQR. However, we have carefully considered the issue of NL-to-CQR translation and it is our belief that implementing these extensions will not present serious problems in this regard [12].

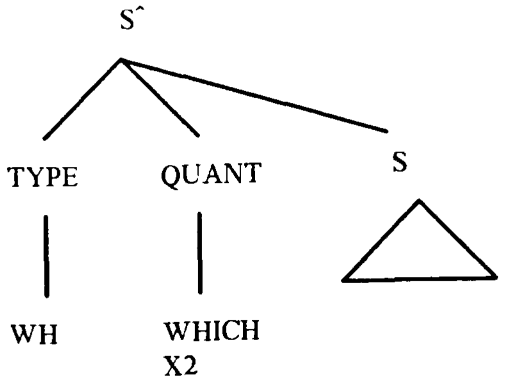
We have extended System X CQR in the following ways:

1. Single Or Multiple Attribute Specification:

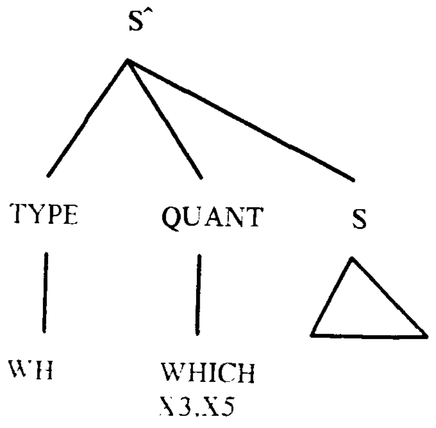
   The quant nodes of the CQR can have either single attributes or multiple attributes specified in the order they should be listed. For instance, the query "Which math majors take a math course ?", has a quant node defined as[13]:

[12] We assume that the various components of the lexicon, the parser and the semantic interpreter can easily be modified to handle our suggested extensions.

[13] The triangle notation under a node replaces a deleted subtree which contains information not relevant to our discussion here.



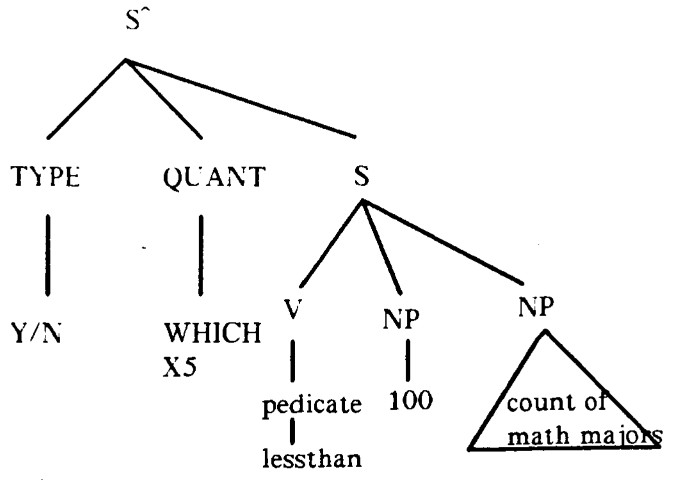
This CQR means that a single attribute X2, which stands for the student numbers of math majors, should be retrieved. Another query, "Display names and grade point averages of computing science students", is represented as follows with its quant node having multiple attributes.



This means that the multiple attributes X3 (which stands for the names of Computing students) and X5 (which stands for the grade point averages of computing students) are to be retrieved.

2. Y/N type combined with WHICH at QUANT node

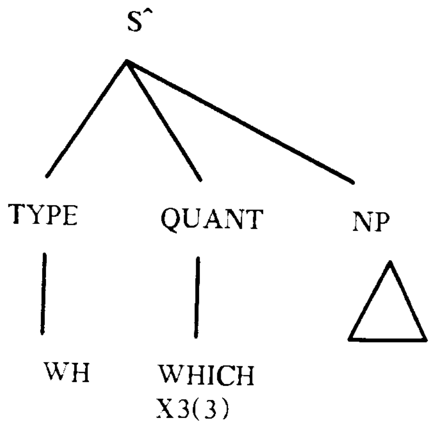
Although the BNF representation of CQR in [McFet 87] allows for a Y/N type to have a WHICH at the QUANT node of a CQR, the current System X implementation does not allow this. For our purposes, we assume that this type of representation is possible. For instance, consider the CQR expression for the query, "Are there at least 100 math majors ?".



Here, X5 stands for the count of all math majors.

3. <u>Handling quota queries:</u>

We provide a second argument for the first variable term of the QUANT node. This allows us to express such queries as: "Get the best three math majors".



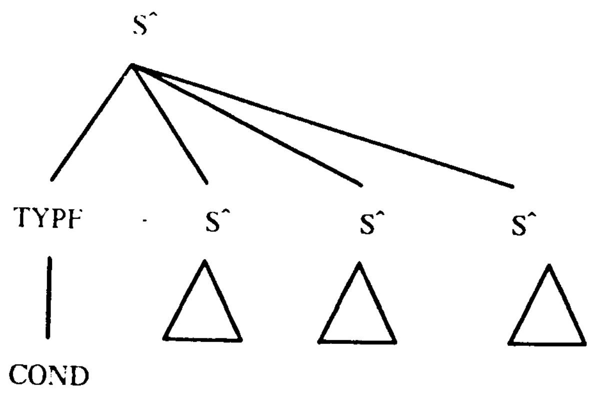
Here, X1 stands for the student numbers of math majors, and the numeral 3 in brackets indicates that three such X1s are to be retrieved.

4. <u>IF-THEN-ELSE Type of Queries</u>

We provide an additional rule for extension of S ^ which is that an S ^ can also expand to <type><S ^ ><S ^ ><S ^ >.

This allows us to represent such queries as:

"If student number 8304 is a math major, give me the math courses he takes else give me his name and department."



The solution to this class of query will be given in Chapter 6.

4.3 The Modified LF

Recall that the CQR is accepted as input by some transformational algorithm to generate a LF. The structure of the System X LF has some resemblance to the system's CQR. In their LF, a query could be any of the following types: a *verification*, a *retrieval*, or an *aggregation*. A verification is a Y/N type of query, a retrieval is a simple retrieval query that does not involve aggregate operations (e.g., "which math majors take all math courses?") and an aggregation is a query that involves some aggregate operations (e.g., "how many math majors take all math courses"). Although the CQR of the system explicitly defines the quantifiers in a query, the representation of universally quantified queries in the LF uses double or nested negations. A LF query usually contains a declaration which is the LF translation of some S or NP branches of the CQR. Further details on the translation of a CQR tree to LF are given in Chapter 6. The BNF grammar for LF used by System X is given in Appendix B [Hall 87].

To enable us handle universally quantified queries procedurally rather than with double or nested negation, and also to cover some classes of queries not yet covered by TQA, we have modified the BNF grammar for LF used by System X. We discuss our modifications next.

In our system's LF, a query can be any of the three types in System X's LF, as well as one of the two new types, a *comparative*, or an *ExplicitStruct*. A comparative query is the same as an "if-then-else" type of query and consists of a verification, followed by two retrievals. An ExplicitStruct represents any query that involves some universal quantification. A query that can be represented by an ExplicitStruct is "which math majors take only math courses?". With ExplicitStruct, we are able to eliminate the use of double or nested negations to represent universal quantification.

Our LF uses *OrderByClause* to specify when a LF declaration should translate to a sequence of SQL statements and an iterative program statement (like FOR loop). This procedure triggered by a LF OrderByClause will become clearer later. Our ExplicitStruct usually contains more than one declaration, and could have an OrderByClause. Each OrderByClause has a *setcondition* which depends on the second quantifier in the English query and its representation in the CQR. If the second quantifier in an English query is ALL, the set condition specified for the OrderByClause is SUPERSETOP, if it is ONLY, the setcondition for the OrderByClause becomes SUBSETOP. At the end of the LF representation for the ExplicitStruct, another setcondition for the very first quantifier in the query is specified. At this level, other quantifiers like WHICH and EXIST are also possible. ONLY at this first level is handled as ALL. The setcondition for WHICH is NOSETOP, that for EXIST is NOTEMPTYSETOP, while that for ALL is SUPERSETOP. An ExplicitStruct LF structure to solve the query "which math majors take only math courses?" is:

```
(
 <LF for set of math majors>
 (ORDERBY math courses
  (SECONDARY math majors)
   (SUBSETOP)
   <LF for set of math majors
     and math courses they take>)
 (NOSETOP)
      )
```

The above ExplicitStruct has two declarations. The first is a simple declaration that represents the set of math majors. The second is an OrderByClause that represents set of math majors order by math courses they take. The setcondition in the OrderByClause is SUBSETOP because of "only" quantifier at the second position in the query while NOSETOP is the setcondition that represents "which" at the first position in the query. The BNF grammar showing only our addition to the System X's LF is given in Appendix C.

4.4 Sample Database Schemas

To enable the reader to understand our references to the database schemas our example queries use, we present the two database schemas used in this thesis.

The first database is the supplier-and-parts database schema discussed in [Date 86a] and presented in Chapter 3 as Fig 3-1.

The second is an academic advice database schema. This is the same schema used by System X. The format of the schema entries is:

RELATION-NAME = key-attr1...key-attr n attribute1...attribute m

for a relation with m non-key attributes and a key composed of n attributes, where $m \geq 0$ and $n \geq 1$. The key attributes are underlined. The academic advice database is given below.

OFFERING = offer# cname semester units

STUDENT = student# name major minor sex status

CLASS = class# offer# sec faculty# text

FACULTY = faculty# name office sex status

APPOINT = faculty# dept

ENROLL = class# student# final-grade

SCHEDULE = group# time room

DEPARTMENT = dept chairman faculty

COURSE = cname description dept

The Academic Advice database scheme

CHAPTER 5

THE TARGET LANGUAGE

This chapter presents definitions of simplified SQL and allowed SQL before discussing the design of our target language. Finally, an example representation of an English language query using our target language is presented.

5.1 Simplified SQL and Allowed SQL

Our target language uses a simplified version of SQL which is described in this Section. The major cause of incomprehensibility in SQL is its use of double and nested negation. In our system, such constructs as:

```
SELECT ------
FROM -------
WHERE NOT [EXIST|IN]
   (SELECT ------
   FROM ------
   WHERE NOT [EXIST|IN]
      (SELECT -------
      FROM ------
      WHERE -------);
```

are excluded from the simplified SQL syntax. Simplified SQL, however, allows SQL constructs involving just one NOT EXIST or NOT IN. Note that the double or nested negation we exclude are those represented at the SQL level and not at the NL level. That is to say that a user is not restricted from posing an English language query with more than one "not" if he desires to. Such queries that introduce double negation at the English language level are translated straight to SQL. If a user decides to introduce incomprehensibility which he could easily avoid by re-phrasing his English query, the system might not bear the responsibility of restricting the number of negations at the SQL level to one. For example, a query like "which suppliers do not supply parts not supplied by S2 ?" is an unnecessary complication. This same query can easily be posed as "which suppliers supply parts supplied by S2 ?". Thus, our target language is concerned with double negations introduced at the SQL level because of universal quantifiers.

To enable us summarise the syntax of our target language, we define *allowed SQL statements*, which are different from simplified SQL. Allowed SQL is TQA-TL, using simplified SQL instead of the regular SQL that TQA-TL normally uses. The syntax of allowed SQL is given below:

Allowed SQL statements:

a. simple retrieval SQL queries without double or nested negation.

   [simple SQL stmt]

b. The Yes/No type of query as defined in TQA-TL

   [POS]

   [simple SQL stmt]

c. TQA-TL's OPERATOR statement that handles quota queries

   [OPERATOR]

   [OPERATOR TYPE]

   [MATH EXPRESSION | VALUE | simple SQL stmt]

   [simple SQL stmt]

   where operator type is one of the following:

   EQUAL, LESSTHAN, MORETHAN, LESSTHANEQ, MORETHANEQ, PERCENT, CHOOSE N, [CHOOSE 1 TO N ---- ORDER BY X DESC].

5.2 Design of Our Target Language

Having defined what an allowed SQL statement is, we now discuss the design of our target language. In forming the target language, we make the syntax as close to that of embedded SQL program (discussed in Chapter 3) as possible. The host language we use is Pascal [SWP 82]. Unlike embedded SQL, our target language uses simplified SQL and allowed SQL statements. Like embedded SQL, the target language permits an allowed SQL statement wherever the host language, Pascal, would allow a Pascal statement.

A statement in the target language can thus be any of the following: an allowed SQL statement, an *assignment statement*, a *whileloop*, an *if statement*, a *repeatloop*, a *print statement*, a *read statement* and a *procedure call*. Those kinds of statements listed above other than allowed SQL statements are formed as defined for the Pascal language. The target language can also take a simple SQL statement as an expression, as described in Section 4.1. Examples of some statements of the target language are:

1. Allowed SQL statement:

```
SELECT A.NAME
FROM   STUDENT A
WHERE  A.NAME='JOHN X'
AND    A.MAJOR='MATH';
```

2. Assignment statement:

```
P:= SELECT A.STUDENT#
    FROM   STUDENT A
    WHERE  A.MAJOR='MATH';
```

3. While Loop:

```
WHILE MORE DO
 BEGIN
   statementlist
 END;
```

4. If statement:

```
IF
  POS
  SELECT A.NAME
  FROM   STUDENT A
  WHERE  A.NAME='JOHN X'
  AND A.MAJOR='MATH';
THEN
  SELECT A.STATUS
  FROM   STUDENT A
  WHERE  A.NAME='JOHN X'
  AND    A.MAJOR='MATH';
ELSE
  SELECT A.MAJOR
  FROM   STUDENT A
```

```
                WHERE  A.NAME='JOHN X';
```

5. Print statement:

```
        WRITE(P);
```

The operators in the target language include set operators like subset ($\subseteq$), superset ($\supseteq$) and non empty set ($\neq \emptyset$). The syntax of the target language is summarized in Appendix B. The target language assumes that all variables have been declared to be the appropriate data type in accordance with Pascal syntax.

5.3 Example Solution of an English Language Query

To demonstrate the use of the target language, we give our target language solution for the same English language query solved in Chapter 3 with embedded SQL and strict SQL. The query is:

"How many suppliers in Paris supply all red screws ?".

```
Q1 := SELECT COUNT (UNIQUE P.P#)
   FROM  P
   WHERE  P.COLOR='RED'
   AND    P.PNAME='SCREW';

{P now contains number of all red screws}

Q2 := SELECT COUNT (UNIQUE SP.P#), S.S#
   FROM  P, S, SP
   WHERE  SP.S#=S.S#
   AND    S.CITY='PARIS'
   AND    SP.P#=P.P#
   AND    P.COLOUR='RED'
   AND    P.PNAME='SCREW'
   GROUP  BY S.S#;

{ Q2 now contains the number of red screws
  supplied by each Paris supplier        }


FOR I := 1 TO number of Paris suppliers do

  BEGIN
    IF Q2 = Q1
    THEN COUNT := COUNT + 1
```

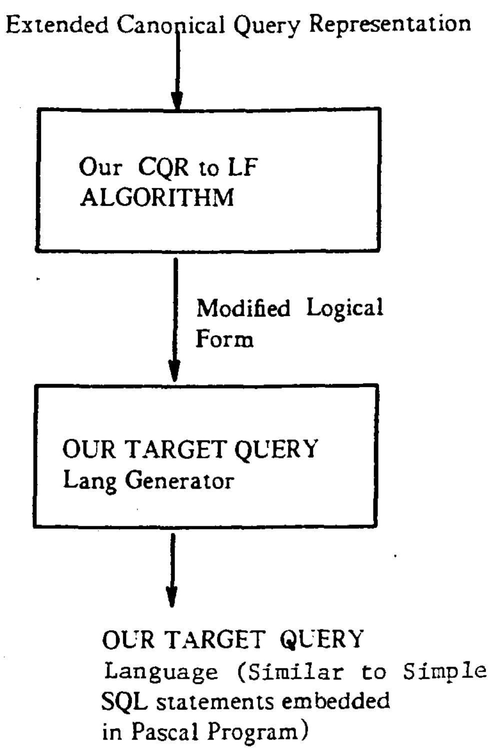

Extended Canonical Query Representation
Our CQR to LF ALGORITHM
Modified Logical Form
OUR TARGET QUERY Lang Generator
OUR TARGET QUERY Language (Similar to Simple SQL statements embedded in Pascal Program)

Fig 5-1: Summary of Part of the System Structure of a NLI Using Our Approach

```
  END;
WRITE (COUNT);
```

This solution is quite similar to the solution with embedded SQL program but quite different from the solution with SQL which used double negation. Our target language program is a concise representation of embedded SQL in Pascal program but with the following differences.

One major difference between embedded SQL program and our target language program is that while embedded SQL program permits double or nested negation, our target language program does not. Secondly, our target language allows some extensions to SQL using the TQA-TL constructs. The use of set operations by our target language only ignores implementaion details. For instance, a set operation $A \subseteq B$, in our target language, becomes a for-loop or a while-loop to check if all elements of array A are contained in array B, when implemented. Although the target language uses Pascal, portability of the target language to systems that prefer other programming languages like PL/1 or C merely requires changing the syntax of Pascal to the syntax of the desired programming language.

What part of the system structure of a NLI (e.g., System X in Fig 2-1) looks like using our solution approach is shown in Fig 5-1.

CHAPTER 6

THE TRANSFORMATIONAL ALGORITHMS

This Chapter discusses our representation of universally quantified queries as well as other classes of queries and presents the transformational algorithms involved. Two algorithms are described. The first algorithm accepts CQR as its input and generates an intermediate form called logical form. The second algorithm accepts the logical form and generates the target language described in Chapter 5 as output. The CQRs, LFs, and our target language representations for eight example queries using these algorithms are given in Appendix E.

6.1 Classification of Quantified Queries

For the purposes of expressing the algorithms, we have classified quantified queries according to the structure of the CQR they generate. We use the information at the Quantifier node of the CQR for this classification. This means that the category to which a query belongs depends on the number of quantifiers involved in the query, and the type of those quantifiers. We allow three quantifiers to be represented in the CQR: ALL <variable>, WHICH <variable> and EXIST <variable>. We restrict the maximum number of quantifiers that can be involved in any one query to three because the meaning of a NL query that involves more than three quantifiers is not readily obvious and it is doubted if any users will ever be interested in posing such a query.

We therefore have three major classes of queries.

1. Queries with one quantifier.
2. Queries with two quantifiers
3. Queries with three quantifiers.

Each major class is further classified according to the type of quantifiers it has and their *order of occurrence* in the CQR. Thus, the three major classes are classified further into the following subclasses. An example of each class of query is presented.

1. Queries with one quantifier.

   a. Quant-type is WHICH <variable>

      For instance, "Which students are taking math100 ?"

   b. Quant-Type is EXIST <variable>

      For instance, "Is John taking math100?"

   c. Quant-Type is ALL <variable>

      For instance, "Does every math major take math100 ?"

2. Queries with two quantifiers.

   We give the orders and combinations of the three defined quantifiers (ALL, EXIST, WHICH) which can currently be represented in System X's CQR.

   a. WHICH <variable1>, ALL <variable2>

      For instance, "Which math majors take all math courses ?"

   b. ALL <variable1>, ALL <variable2>

      For instance,

      1. "Do all math majors take all math courses ?"

      2. "Do all math majors take only math344 ?" [14]

   c. EXIST <variable1>, ALL <variable2>

      For instance, "Was a math course taken by every student ?"

   d. EXIST <variable1>, EXIST <variable2>

      For instance, "did John not take every math course ?" [15]

Some negated universally quantified NL queries are interpreted at the CQR level with existential quantifiers (EXIST's), and such queries account for the class which contain more than one EXIST quantifier in the CQR structure of a query. For instance,

[14] This query is in this subclass because "only" is represented as ALL at the CQR level.

[15] Note that some other combinations that may not be possible at the current stage of the System X prototype are not represented.

the query: "Did John not take every math course ?" is interpreted in CQR as "Does there exist a student named John and does there exist a math course such that John did not take the course ?".

3. Queries with three quantifiers.

   We can have the following sub-categories:

   a. WHICH <variable1>, ALL <variable2>, ALL <variable3>

      For instance, "which students take every course from every instructor?"

   b. EXIST <variable1>, ALL <variable2>, ALL <variable3>

      For instance,"was a math course taken by every student in every department ?"

   c. ALL <variable1>,ALL <variable2>, ALL <variable3>

      For instance, "Do all math majors take every course from every instructor ?"

Other classes of English language queries have CQR's that belong to one of these major and sub categories. For instance, quantified queries with COUNT fall into the groups with WHICH and ALL's or WHICH and EXIST's. Universally quantified queries that contain English "only" are also represented in terms of ALL.

6.2 Idea Behind Our Approach

To avoid using double or nested negation to simulate universal quantification, we break down the procedure into simple step-by-step retrievals. This Section also explains how the algorithms of Sections 6.4 and 6.6 handle all the classes of queries outlined in Section 6.1. The order in which quantifiers appear in the CQR of an English query is the same as the order in which they appear in the input query. Four quantifiers, WHICH, EXIST, ALL, and ONLY are of primary concern to us in this thesis, although the target language has been designed to handle many more. The following rules are applicable:

* Each of the four quantifiers has an associated set operator. A WHICH quantifier at the CQR

has the set condition NOSETOP attached to it, which means that there is no set operation involved at the level the quantifier appears. An EXIST quantifier has the set condition NOTEMPTYSETOP attached to it, which means that at the level it appears there is a test for a non empty set ($\neq \emptyset$). An ALL quantifier has the set condition SUPERSETOP attached to it which means that there is a test for a superset operation between some two sets. An ONLY has a set condition SUBSETOP attached to it which means that there is a test for a subset operation between some two sets.

* In our algorithm, the processing of ONLY when in the first quantifier position is the same as the processing of ALL in the same position. However, the set operator that applies to objects and subjects in the set of data retrieved changes when ONLY is in the second quantifier position.

* We start solving a query with three quantifiers as though it is a query with just two quantifiers, its first and second quantifiers. Secondly, we collapse the part of the query containing the two innermost quantifiers to get a set of values. In this case, two interpretations are possible. For instance, with the query "which math majors take all math courses from all math instructors?", the first interpretation looks for those students who take all courses taught by all math instructors. The second interpretation wants each of the courses taken by the students to be taught by all math instructors. The algorithm has to be informed by the CQR which interpretation is desired so that the appropriate rule in our algorithm is triggered. If the first interpretation is desired, the part of the query containing the two innermost quantifiers is translated to a simple SQL query to get the desired set at the second step. In the above example the set obtained from collapsing the part of query containing the two innermost quantifiers is {courses with instructors in math dept}. If the second interpretation is desired, then the part of the query containing the second and the third quantifiers is first translated as though it is a complete query containing the two quantifiers WHICH and the third quantifier in that order. For instance, the query "do all math majors take all math courses from all instructors?" is represented at the CQR as ALL

(math majors), ALL (math courses), ALL (instructors). To get the desired set from collapsing the two innermost quantifiers, we look at that part of the query as WHICH (math courses), ALL (instructors). Then we pass this complete query structure to our algorithm that handles two quantifiers which returns the {math courses each of which is taught by all math instructor}. Note that ONLY in the third position has just the second interpretation as the first interpretation does not make sense.

*   The solution for ONLY when it is the first quantifier is the same as that for ALL because of the structure of the CQR.

Our general solution for queries is outlined below. This algorithm describes the criteria that our CF-to-LF and LF-to-Target-language algorithms are based on, but does not give details about any of these algorithms. The detailed discussions of our two algorithms are presented in the following Sections.

1. If the query has only one quantifier and it is an EXIST or a WHICH, it is solved in the way TQA and System X would solve it. If the only quantifier is an ALL <subjects>, the algorithm proceeds as follows [16]:

   Step 1: A := {subjects}

   Step 2: B := {subjects satisfying the conditions shown on the CQR entire S branch}

   Step 3: If B $\supseteq$ A then say "yes" else say "no";

   Example 1.1

   To answer the query, "do only math majors take math100?", we proceed as follows. This query is represented at the CQR as "are all students taking math100, math majors?".

   Step 1: A := {students taking math100}

   Step 2: B := {math majors taking math100}

[16] The set notation { } means set of all objects described by what is inside the brackets. For example, {student#} means the set of all student numbers.

Step 3: If B $\supseteq$ A then say "yes" else say "no";

2. If the query has two quantifiers, the query structure is:

Quant1 <subject>, Quant2 <object>

where Quant1 can be EXIST, WHICH, or ALL, and Quant2 can be EXIST, ALL, or (implicitly) ONLY. Our general solution here is:

```
Step 1: A := {subjects}
   {This step 1 is omitted if Quant1 is WHICH or EXIST.}
Step 2: B := {objects}
Step 3: C := {objects grouped by subject}
   For i := 1 to number of subjects
      If {object for ith subject} <setcondition for Quant2> B
         then make subject i an element of D.
end;
Step 4 :
   (a)If the setcondition for Quant1 is NOSETOP then write D;
   (b) otherwise If the set condition for Quant1 is NOTEMPTYSETOP then write ('( If
   D ≠ Ø then say "yes" else say "no")');
   (c) otherwise write ('(If D <setcondition for Quant1> A then say "yes" else say
   "no")');
      {i.e, if Quant1 is ALL, setcondition is SUPERSETOP}
```

<u>Example</u> <u>2.1</u>

To answer the query, "do all math majors take all math courses ? ", we proceed as follows:

Step 1: A := {math majors}

Step 2: B := {math courses}

```
Step 3: C := {<math courses(mc)> grouped by math majors (mm)}
          For i := 1 to number of math majors
             If {mc of ith mm} $\supseteq$ B
                then make math major i an element of D.
          end;
Step 4: If D $\supseteq$ A then say "yes", otherwise say "no".
```

In this case, our LF will specify SUPERSETOP both in the OrderByClause and after the OrderByClause because the second quantifier is ALL and the first quantifier is also ALL.

Example 2.2

To answer the query, "do all math majors take only math courses ?", the step by step solution proceeds as follows:

```
Step 1: A := {math majors}
Step 2: B := {math courses}
Step 3: C1:= {<courses(c)> grouped by math majors(mm) }
          For i := 1 to number of math majors
             If {c of the ith math major} $\subseteq$ B
                then make math major i the next element of D
          end;
Step 4: If D $\supseteq$ A then say "yes" otherwise say "no";
```

In this case, our LF will specify SUPERSETOP in the OrderByClause and SUBSETOP after the OrderByClause for the final set operation.

Example 2.3

To answer the query, "Which math majors take all math courses ? ", we proceed as follows:

```
Step 1: A := {math courses}
Step 2: B := {<math courses(mc)> grouped by math majors (mm)}
        For i := 1 to number of math majors
          If {mc for the ith mm} $\supseteq$ A
            then make math major i an element of C.
        end;
Step 3: Write C;
```

In this case, our LF will specify NOSETOP as the setcondition for the final set operation and SUPERSETOP for the OrderByClause because of the two quantifiers WHICH and ALL involved in the query.

Example 2.4

If the query is "is there a math major taking all math courses ?", the solution is:

```
Step 1: A := {math courses}
Step 2: B := {<math courses(mc)> grouped by math majors (mm)}
        For i := 1 to number of math majors
          If {mc for the ith mm} $\supseteq$ A
            then make math major i an element of C.
        end;
Step 3: If C $\neq \emptyset$ then say "yes" else say "no";
```

3. If the query has three quantifiers, the first step is to look at the query as a complete query with just two quantifiers, its first and second quantifiers. The second step is to collapse the two innermost quantifiers to get a set. This last step replaces step 2 of algorithm 2 above. Other steps proceed as for case of two quantifiers (first and second) which we started with. The structure of this class of query is:

Quant1 <subject>, Quant2 <object1>, Quant3 <object2>

We start solving the query as though it is:

Quant1 <subject>, Quant2 <object1>

```
Step 1: A := {subjects}
  {This step 1 is omitted if Quant1 is WHICH or EXIST.}
Step 2: B := {object1 by Quant3 object2}
  {Step 2 is the set obtained by collapsing the two innermost quants}
Step 3: C := {object1 grouped by subject}
  For i := 1 to number of subjects
    If {object1 for ith subject} <setcondition for Quant2> B
      then make subject i an element of D.
  end;
  Step 4 :
  (a)If the setcondition for Quant1 is NOSETOP then write D;
  (b) otherwise If the set condition for Quant1 is NOTEMPTYSETOP then write ('( If $D \neq \emptyset$ then say "yes" else say "no")');
  (c) otherwise write ('(If D <setcondition for Quant1> A then say "yes" else say "no")');
    {i.e, if Quant1 is ALL, setcondition is SUPERSETOP}
```

Example 3.1

Consider the answer to the query "Which math majors take all courses from all math instructors ?" [17]

[17] Note that in solving three quantifiers using the first interpretation, we take it that the element of object1 counts just once, in this case, a course counts just once. That is to say, if there are just two instructors and both instructors teach a particular course, a student needs to take that course just once and not from both instructors. For instance, if instructor 1 teaches math101 and math 201 and instructor 2 teaches math201 and math301, a student who takes math101, math201 and math301 is seen as having taken all courses from all math instructors.

```
Step 1: omitted because Quant1 is WHICH
Step 2: A := {courses taught by all math instructors}
Step 3: B := {<courses(c)> grouped by math majors(mm)}
  For i := 1 to number of math majors
    If {courses for ith mm} $\supseteq$ A
      then make math major i an element of C.
  end;
Step 4: Write C;
```

6.3 Description of Procedures in CQR-to-LF Algorithm

This Section describes the procedures used in the algorithm that translates CQR to LF. The detailed algorithm that uses these procedures is given in Section 6.4.

1. Procedure Translate-Tree (S ^).

The CQR of every NL query has a tree structure that looks like one of the following: S ^ --> TYPE, Quantifier, S; [18] S ^ --> TYPE, Quantifier, NP; or S ^ --> TYPE, S ^ , S ^ , S ^ for comparative queries, where S ^ , TYPE, Quantifier, S, and NP are all nodes of the CQR tree. The procedure Translate-Tree (S ^) is a recursive procedure that takes an S ^ node as its input and generates the LF that conforms to our modified LF grammar. The procedure starts by writing the query type in the LF. Then, it traverses the quantifier nodes from left to right, processing each quantifier and its object in turn. When it encounters a quant, it invokes the procedure Translate-Subtree to process it. After all the quants have been processed, the rest of the S node next to the quantifier node is translated to LF only if there is just one quantifier in the quantifier node. Then, the set operation to apply in the query structure is specified. The set operation specified at this level depends on what the first

[18] A notation "S ^ -->TYPE,Quantifier,S" means that node S ^ has descendants TYPE, Quantifier and S from left to right in the tree.

quantifier in a query is. For instance, the query "Do all math majors take only math courses ?" has ALL as the first quant, while the query "Which math majors take all math courses?" has WHICH as the first quant. If the first quantifier is a WHICH, the set operation specified is NOSETOP. If the first quantifier is an ALL, the set operation specified is SUPERSETOP. If the first quantifier is an EXIST, the set operation specified is NOTEMPTYSETOP.

2. Procedure Translate-Subtree (subtree,quant)

This procedure takes a quantifier and the *object* of the quantifier as its input, processes that quantifier based on the position of the quantifier and its type in the CQR, to produce some LF as it output.

The main procedure passes as parameters to this subroutine, the quantifier and the object of that quant. The object of a quantifier in the CQR is the subtree that has the quantifier variable as its index. For instance, in Appendix E, the CQR for Example query 1, the object of the quantifier 'EXIST X1' is the NOM node whose attribute name is cname in the relation offering, and this NOM node has descendants NOM and S. The index of the descendant NOM is X1 which identifies this object in the CQR tree. Every other reference to this object, like translate to LF the object of X1 or delete the object of X1, refers to the S node following this NOM. The CQR also allows for another type of NOM whose descendants are NOM and S ^. If the index of a NOM with this structure is encountered, the object of that index is the subtree for S ^, which is a complete query. The object of X1 in the CQR of Example Query 1 in Appendix E, is shown as Fig 6-1. Subroutine Translate-Subtree, when a quantifier and its object are passed in, checks the type of quant. If the quantifier is for universal quantification (an ALL or an EXIST with negation as the first V node), and is the first ALL in the query, the subroutine translates the object of this quantifier to LF. But if it is not the first ALL in the query, it will in addition open up an OrderByClause using the variables of the quantifier and that of the previous quant. It specifies the OrderByClause in such a way that the attribute in the database that is bound to the current variable should be

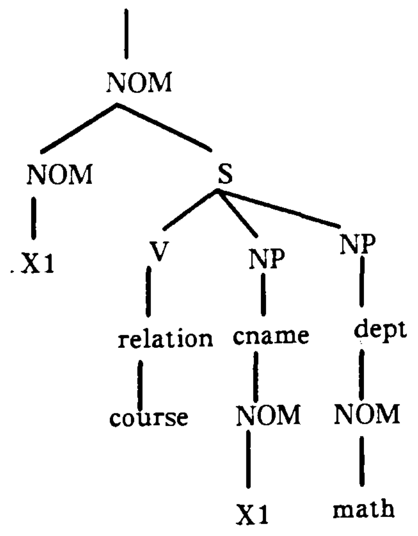


Fig 6-1: The Object of the Quant (EXIST X1) From Example 1 in Appendix E.

selected, but ordered by the attribute bound to the previous variable. For instance, if the query is "do all math majors take all math courses ?", the quantifier node of the CQR will have a structure that looks like: ALL X1, ALL X2. Here, X1 is bound to math majors and X2 is bound to math courses. When processing the second quantifier ALL X2, the opening of the OrderByClause specified is (ORDERBY X1 (SECONDARY X2). This means get the set of math courses taken by each math major with the rest of the OrderByClause specified. Next, the algorithm specifies the set operator to be used inside the iterative statement in the target language triggered off by the OrderByClause. If the universal quantifier is ALL as in the above query, the user is interested in a math major only if the set of math courses he has taken contains all available math courses. Thus, for ALL, the algorithm will specify SUPERSETOP. If the universal quantifier involved is only, as in "do all math majors take only math courses ?", it will specify SUBSETOP. Next in the OrderByClause, the algorithm translates to LF the subtree linking the current subtree to the previous subtree. If the quantifier is WHICH, it checks whether an aggregate operator is involved, as in the query "how many math majors take all math courses ?". If there is an aggregate operator it takes note of that and the variable bound to the result of the operation. The algorithm also takes note if a WHICH quantifier has a numeral argument to represent a quota query. If there is a WHICH in the quantifier node and at least one ALL, then the aggregate operator is specified just before the OrderByClause.

3. Procedure Translate-CQR-Subtree-to-LF.

This procedure is called by the Translate-Subtree procedure discussed in 2 above. This procedure translates a subtree passed in as a parameter into LF, using the same technique used by TQA and System X.

In these two systems, when a CQR subtree is to be translated to LF, each variable appearing in the branch or branches to be translated must be declared exactly once. If a variable is to hold aggregate values like sum of, count of, etc., it may be declared using BAGX. BAGX is used when we do not want to discard duplicate values in the "bag" of values being summed etc. Otherwise variables are declared using SETX. After declaring the variables, if there is more than one relation involved in the subtree, the condition "AND" is specified. Then the relation clauses are specified. A relation clause is formed out of an S node if the V branch of S is of type relation. The relation clause first shows the relation name, then the attribute names from that relation and then the indices of these branches. The indices can either denote attribute values, or variable names to be bound to these attribute values. Shared variables[19] in the CQR structure specify the join path (the connecting link between one relation and another).

This procedure specifies other clauses like *comparison* or *setcondition* , which are terms used in the BNF description of the LF grammar. A comparison is of the form "(<compop> <variable> <attrvalue>)". For example, (GREATERTHAN X3 3) is a condition that requires that value bound to X3 be greater than 3. This comparison is formed if a V node found in a CQR subtree is a comparison predicate (e.g.,EQUAL, etc.). The setcondition is a

[19] Variables that are indices to more than one S node of the CQR.

new operator not used by TQA and System X, but which is necessary for our program-like set operations. In our algorithm, the set condition could be specified more than once in a LF query structure. Every OrderByClause in the LF should have a set condition specified. The four set conditions used, as described earlier, are SUBSETOP, SUPERSETOP, NOTEMPTYSETOP and NOSETOP standing for subset, superset, not an empty set and no set operations, respectively. Once a set operation is specified, the query is known not to be a simple retrieval query. Thus in our approach, the LF structure is an ExplicitStruct, (that is, the LF structure has more than one declaration). A LF declaration can stand for a complete query. From information found in the CQR, we are able to specify which of the setconditions is required.

The translation of the subtree shown in Fig 6-1 to LF will give the following:

```
(SETX 'X1
  (RELATION OFFERING
   (CNAME DEPT)
   (X1   'MATH)
   (=    =)))
```

6.4 CQR to LF Algorithm

This Section presents the algorithm that translates a CQR to LF, and uses the subroutines described in Section 6.3 above.

Procedure Translate-Tree (S ^);

1. IF Query type of S ^ is COND

```
1.1 THEN do the following:
    1.11 Let S1^, S2^, S3^ be the descendant nodes of S^ from left to right.
    1.12 write ('(COND* ')[20];
    1.13 Translate tree (S1^);
    1.14 Translate tree (S2^);
    1.15 Translate tree (S3^);
1.2 otherwise do the following:
    1.21 IF Query type of S^ is Y/N
    THEN write ('(INT* ');
    1.22 IF Query type of S^ is WH
    THEN write ('(');

    1.23 Initialize variables
        Number-of-quantifier = 0;
        Only = 0;
        Whichflag = 0;
        Existflag = 0;
        Allflag = 0;
        Previous-subtree = null;
        Var = null
        Agg-op = null
        Agg-op-var = null
        Whichalls = 0
        Threequantifier = 0
        Reading = 0
```

[20] The * in some of the key words in the LF is just a notation and has nothing to do with the Kleene closure

Thirdquantifier = null

{ Number-of-quant keeps count of the number of quants in the quantifier node. Only becomes 1 if the algorithm discovers from the search that the universal quantifier at the second level involved in the particular query is ONLY. Whichflag is set to 1 when the first quantifier node is WHICH and Existflag is set to 1 when the first quantifier is EXIST. Allflag is set to 1 when the first quantifier is an ALL Reading is used to indicate which interpretation should be in effect for queries with more than one interpretation. Previous-subtree stores the name of the variable which is the index to the previous subtree processed. Var contains the variable to a WHICH quant. If the WHICH variable is defining an aggregation in the CQR like (quantity), Var is made to contain the object that this aggregation should be applied to. Agg-op is used to remember the aggregate operator (e.g, TOTAL) in a CQR. Agg-op-var is used to remember the CQR variable bounded to the result of an aggregate operation. Whichalls and Existalls are set to 1 only if the quantifier node has the structure that looks like "WHICH X1, ALL X2 .." or "EXIST X1, ALL X2 .." respectively; after processing the first ALL. Threequant is set to 1 if there are three quants in the quantifier node. Thirdquant holds the third quantifier in a three quantifier structure. }

1.24 Search for the quantifier node.

1.25 IF there are three quants in the quantifier node then do the following:

(a) Threequantifier = 1

(b) Thirdquantifier = the third quantifier

(c) Reading = the desired interpretaion (1 or 2)

1.26 For every quantifier (q) up to the second one in the quantifier node, do the following:

    1.261 Increase the Number-of-quantifier by 1.
    1.262 IF Number-of-quantifier is 1 then do:
        (a) IF quantifier is WHICH, set Whichflag to 1
        (b) IF quantifier is EXIST and the V node of S is not an negation operator,
        then set Existflag to 1.
        (c) IF quantifier is ALL or (EXIST and the V node of S is not an operator),
        then set Allflag to one.

        1.263 Translate-Subtree (object,q)

        { This is a procedure that translates the object of the quant, which is a subtree of the CQR that has the quantifier variable as its index }

1.27 IF Number-of-quantifier = 1 then Translate subtree (proposition) to LF.

{Some subtrees that have already been processed have been deleted from the CQR for the query. The deletion of processed subtrees occurs only if the query is simple retrieval type. After processing all quants, the branches of the main CQR that are left are translated as the proposition.}

1.28 (a) IF Allflag = 1

THEN write ('(SUPERSETOP)');

{ This means that the query contains the quantifier ALL in the first position (e.g, "Do all math majors take only math courses?"}

(b) IF Whichflag = 1

THEN write ('(NOSETOP)');

(c) IF Existflag = 1 and Number-of-quantifier > 1

```
    THEN write ('(NOTEMPTYSETOP)');

    1.29 End the translation of the CQR to LF by inserting a closing bracket.

Procedure Translate-Subtree (subtree,quant);
1.  IF quantifier is ALL or EXIST with the first V node equal to negation

  {This allows the processing of any ALL quantifier in the quantifier node; if it is
  EXIST with the first V node as negation, it looks at it as an ALL .}
  1.1 THEN do the following :
    1.11 IF Allflag = 1 and Previous-subtree is not null and is nested in the present
    subtree, then Only = 1;
    1.12 IF quantifier is the first ALL quantifier
    {It can use the variables Existflag, Existalls and Whichflag and Whichalls to find
    out when the first ALL quantifier is being processed. }
    1.121 then translate subtree to LF.
    1.122 otherwise do the following:
      1.1221 (a) If Only = 0 then just translate the subtree to LF without links to
      other subtrees.
      1.1221 (b) If Only = 1 then translate the CQR subtree adjacent to that of the
      subtree to LF.
      1.1221 (c) If Threequants = 1 and Reading = 1 then translate subtree linking it
      to the subtree for Thirdquant.
      1.1221 (d) If Threequants = 1 and Reading = 2 then CQR-to-LF (WHICH
      variable2, Thirdquantifier variable3)
      1.1222 IF the V node of the first S is negation, then write ('(NOT* ');
      1.1223 Open OrderByClause as (ORDERBY Previous-subtree (SECONDARY
        variable)
        {variable is the variable arguement to quant}
        1.1224 IF Only = 1
            (a) then write ('(SUBSETOP)');
            (b) otherwise write ('(SUPERSETOP)');
        1.1225 Translate the subtree to LF linking to only elements of the previous
        subtree.

2.  IF quantifier = WHICH variable,
    2.1 then do the following:
        (a) if the V node of the first S is an operator, set Var to the next variable in the
        CQR after the operator node, traversing the tree left to right. Set Agg-op-var to
        the quantifier variable, and set Agg-op to the Aggregate operator in the CQR.

        (b) otherwise set Var equal to the quantifier variable.

3.  If it is the second quantifier and Whichflag = 1 then do the following:
    3.1 IF Agg-op ~= null then write ('(Agg-op Agg-op-var)');
    3.2 Open OrderByClause as (ORDERBY Var (SECONDARY Previous-subtree)
    3.3 IF Only = 1,
            (a) then write ('(SUBSETOP)')
            (b) otherwise write ('(SUPERSETOP)');
        3.4 Translate the subtree for Var to LF linking it to elements of the previous
        subtree.

4.  IF it is the second quantifier and Existflag = 1
    4.1 Open OrderByClause as (ORDERBY variable1 (SECONDARY Previous-subtree)
    4.2 IF subtree is nested in subtree for Thirdquantifier
```

```
        (a) then write ('(SUBSETOP)');
        (b) otherwise write ('(SUPERSETOP)');
    4.3 Translate subtree for Var to LF linking it to elements of previous subtree.
5.  Previous-subtree = subtree
6.  Return
```

6.5 Description of Procedures Used in LF to Target Language Algorithm

This Section describes the procedures that are used to translate LF to the target language. The detailed algorithm that calls these procedures is given in Section 6.6.

1. Procedure Translate-LF (LF)

This is a recursive procedure that takes a LF tree as its input and generates the target language, which is a program-like structure with some embedded simplified SQL. If the LF input is that of a conditional query, the algorithm inserts an "IF" before the SQL translation of the first LF query structure. A "THEN" is inserted before the translation of the second LF query structure and an "ELSE" is inserted before the translation of the third LF query structure. However, if the query structure is a simple retrieval query, it just translates the query to SQL using the same technique that TQA and System X use (by "simple retrieval query" we mean a query that does not involve universal quantification). The simple retrieval query can be a verification like "is John a math major ?", or a retrieval like "which math majors take math100 ?", or an aggregation like "how many math majors take math100?". When a query is not a conditional type and is not a simple retrieval, in our current model, it is an ExplicitStruct. These are quantified queries like "do all math majors take all math courses ?".

If the LF structure is that of an ExplicitStruct, the algorithm checks to see if the query is a negative query, with "(NOT* " early up in the LF tree, or if it contains an aggregate operator like "TOTAL", and it takes note of which of these conditions is/are true. Then, the algorithm translates each declaration, (which can be a complete LF for a simple SQL structure) into SQL using the same technique used by TQA and System X. The SQL form of each declaration is made the right hand side of a program assignment statement that has a unique variable as its left side. The unique variables used in the program are pushed onto a stack, and popped off for subsequent set operations. Then if the LF for the ExplicitStruct contains a setcondition with value "SUPERSETOP", the last unique variable in the stack is popped off and written and the set operator to follow is $\supseteq$ (i.e, superset) and the next unique variable is popped off stack and written. If the setcondition is "NOSETOP", the algorithm pops off just the last unique variable and writes it as the result of the query; no set operation takes place in this case. If the setcondition is SUBSETOP, the algorithm pops off the last unique variable from the stack and writes it, then specifies the set operator $\subseteq$ (i.e, subset) before it pops off the next unique variable and writes it. If the LF has no negation at the beginning, then the result of the query becomes the result of these last set operations. However, if negation is involved, the result of the query becomes the reverse of the result of the last set operation. If there is an aggregate operator involved in the LF, a procedure to evaluate the aggregate operation and write the result is invoked.

2. Translate to SQL (declaration)

Recall that the LF translation of CQR shown in Fig 6-1 is:

```
(SETX 'X1
  (RELATION OFFERING
   (CNAME DEPT)
   (X1  'MATH)
```

(=   =)))

This is an example of a LF declaration. To translate a declaration to SQL, the attribute(s) to be specified in the SQL SELECT clause is/are that bound to the variable(s) in the first SETX or BAGX in a LF declaration; in the above case, OFFERING.CNAME. In our approach, to allow multiple attribute retrieval, the first SETX declares all the variables bound to the attributes to be retrieved. Other variables are declared in subsequent SETX statements. And the relations to be listed in the FROM part of the SQL query are all relations in the LF relation clauses of the LF declaration; in the above case, only one relation (OFFERING) is involved. The WHERE part of the SQL query uses the links between relations in the LF to specify join paths and also values bound to attributes as well as condition clauses. The complete SQL translation of the above LF declaration is:

```
SELECT A.CNAME
FROM  OFFERING
WHERE  A.DEPT = 'MATH';
```

3. Procedure Operator-Evaluate (variablename, Agg-op)

This procedure applies the aggregate operator involved in the query to the program variable passed in as argument, which at the time of invocation has some values. For instance, if the aggregate operator is "TOTAL" and the variable is V1. V1 at the point of invocation has a list of say, students. So, the procedure Operator-Evaluate will return the number of elements in V1 as the result.

6.6 LF to Target Language Algorithm

This Section presents the algorithm that translates the LF of the query into the target language. It uses the procedures described in Section 6.5.

```
Procedure Translate-LF (LF);

1.    IF LF Query is <comparative>

    1.1 THEN do the following:

        1.11 Let LF1, LF2, and LF3 be the <verification> <retrieval> and
        <retrieval>parts of the query.

        1.12 write ('IF');

        1.13 Translate_LF (LF1);

        1.14 write ('THEN');

        1.15 Translate_LF (LF2);

        1.16 write ('ELSE');

        1.17 Translate_LF (LF3);

    1.2 otherwise do the following:

        1.21 Initialize variables

        1.22 IF query is <verification> or <retrieval> or <aggregation>, then translate
        query to SQL using the same technique as TQA and System X.

        {In any of these three query structures, only simple SQL statements that do not
        involve double or nested negations are involved }.

        1.23 IF Query is <ExplicitStruct>, then do the following:

          1.231 IF there is a (NOT* preceeding the first <declaration>, then set the
          program variable Negate to 1.

          1.232 IF there is an aggregate operator in the LF set the program variable
          Aggop to the the aggregate operator.

          {aggregate operators in the LF are such terms as TOTAL, QUANTITY and
          AVERAGE }
```

1.233 If there is a numeral operator in the LF, set the program variable, Numeral to the value.

1.234 For every declaration (d) in the LF, do the following :

```
    1.2341 Form a unique variable V.
    1.2342 write (':=');
    1.2343 If (d) is not an OrderByClause then do the following;
        (i) Translate to SQL (d);
        (ii) Push V onto a stack.
    1.2344 If (d) is an OrderByClause then do the following:
        (i) Create a view called Temp as (Translate to SQL(d))
```

{If the declaration is an OrderByClause, then the first action is to create a view (a virtual relation) that has attributes bound to the SECONDARY variable and the ORDERBY variable of the OrderByClause. For instance, assume the OrderByClause contains (ORDERBY X2 (SECONDARY X4) ..) and X2 is bound to CNAME while X4 is bound to STUDENT#. The SQL view creation construct here is

```
CREATE VIEW TEMP AS
SELECT STUDENT#, CNAME
FROM (some realations as found in the LF)
WHERE (conditions in the LF hold);
```

Note that the condition ORDER BY CNAME, does not have to be specified. We use it in the algorithm just for purposes of clarity.}

```
        (ii) write ('TEMPVAR :=');
        (iii) write SQL construct to select distinct elements of the attribute bound
        to the SECONDARY variable of the OrderByClause, from the view temp.
```

(iv) Open up a Forloop with index going from one to the number of elements in Tempvar.

(v) write ('SETi := ');

(vi)write the SQL construct to select the attribute bound to the ORDERBY variable from the view temp, where the attribute bound to the SECONDARY variable is the indexed element of Tempvar.

(vii) Make a unique variable V.

(viii) write ('IF SETi <setcondition> pop(stack)

then make the ith element of Tempvar the next element of the last unique variable.

(ix) Close the Forloop.

(x) Push the unique variable onto the stack.

1.234 (a) IF the next <setcondition> = SUPERSETOP then IF Negate = 0

then write ('IF pop(stack) $\supseteq$ pop(stack)

THEN write ("yes");

ELSE write ("no");');

otherwise write ('IF pop(stack) $\supseteq$ pop(stack)

THEN write ("no");

ELSE write ("yes");');

(b) IF there is a <setcondition> = NOSETOP then IF Agg-op = null

(i) If Numeral=0 then write ('write pop(stack);');

(ii) If Numeral $\neq$ 0 then write only Numeral elements of pop(stack);

otherwise if Agg-op $\neq$ null then write ('write Operator-Evaluate(pop(stack),Agg-op)');

(c) IF there is <setcondition> = SUBSETOP then IF Negate = 0

then write ('IF pop(stack) $\subseteq$ pop(stack)

```
        THEN write ("yes");
        ELSE write ("no");');
    otherwise write ('IF pop(stack) ⊆ pop(stack)
        THEN write ("no");
        ELSE write ("yes");');
(d) IF there is <setcondition> = NOTEMPTYSETOP then IF Negate = 0
    then write ('IF pop(stack) ≠ ∅
        THEN write ("yes");
        ELSE write ("no");');
    otherwise write ('IF pop(stack) ≠ ∅
        THEN write ("no");
        ELSE write ("yes");');
```

CHAPTER 7

CONCLUSIONS

We have demonstrated that the SQL query language, though powerful and popular, is not an ideal target language for NLIs because:

* SQL is quite restrictive and can not express many common NL queries some of which are quota queries, conditional queries and anaphoric queries.

* SQL uses double and nested negations to simulate universal quantifiers and this makes SQL translations of such NL queries hard to understand leading to lack of user confidence in the translations.

Secondly, we have described how the TQA system extends SQL to cover more classes of English language queries than SQL. However, the TQA target language is also not very ideal for NLIs because:

* TQA uses the full SQL syntax, thus inheriting all the hard-to-understand features of SQL.

* Many important classes of English language queries are still not handled by TQA system. TQA creates an additional external construct for each major class of English language queries, e.g., POS for interrogative queries, and so on. There may never be an end to the number of such external constructs that need to be created to handle many classes of English queries. That means the TQA target language is not very systematic.

Thirdly, we have argued that mapping NL queries to simplified SQL embedded in a host language program, rather than directly to SQL or its extension, has the following advantages:

* will increase the capabilities of the NL system

* remove the incomprehensibility posed by some of the constructs in the full SQL syntax.

* can allow intermediate result of a query to be stored for future processing of such queries as anaphoric queries.

* although the host language for our program-like target language is Pascal, portability of the

target language to systems that prefer other host languages like PL/1 or C is easy and requires merely changing Pascal syntax to the syntax of the new host language.

In line with this argument, we have designed a target language which uses only simplified SQL syntax embedded in a host language program. Our target language also allows some of TQA's extension of SQL in the host language program as well as set operators. Our definition of simplified SQL is SQL syntax that excludes all constructs involving double or nested negation. Constructs involving a single negation are allowed in simplified SQL.

Fourthly, we developed an algorithm to translate the CQR of the natural language query to our logical form, which is an extension of TQA's and System X's logical form. We also developed a second algorithm that translates logical form to our target language. Our algorithms are specially designed to handle universally quantified queries and to remove incomprehensibility.

In the remainder of this chapter, we discuss some promising approaches to extending this work.

7.1 Observations and Future Research

The observations made in this section are speculative and require further investigation. However, they indicate what appears to be a natural and promising extension of the work reported in the thesis thus far.

7.1.1 Number Of Universal Quantifiers in a Query

This thesis is restricted only to queries with a maximum of three quantifiers. However, if it is necessary to pose NL queries that contain more than three quantifiers, we think that our work can be extended to accommodate them. We also think, however, that the meaning of NL queries that involve more than three quantifiers is not obvious, and that users might want to avoid such naturally confusing queries.

Example: "Does every female student in every math course take every course from every math instructor ?".

7.1.2 Query and Search Optimization

It was not the objective of this thesis to focus on the response time of the queries. However, this is a factor that needs some design attention. At present, there are several parses of our input trees and we believe that techniques could be developed to optimize the query and also reduce number of parses. For instance, Example Query 6 in Appendix E, has a step that could be optimized to save time. Since, the only element of V2 in that example is identified by itself and is MATH344, it is not necessary to issue the SQL command to pick this single element identified by itself. However, since we are using a general procedure to handle all cases, it is outside the scope of this thesis to consider query and search optimization.

7.1.3 Future Implementation of Our Algorithms

The possibility of implementing our algorithms in the future by probably modifying the existing modules of System X has always been given serious consideration during the design. Our first algorithm takes as its input the CQR generated from the upper modules of System X. The modules of System X that need to be extended are those shown in Fig. 5-1.

Little modification is needed to get our target language conform precisely to the syntax of the SQL programming language interface available in the database management system (DBMS) available. Some of these modifications are in defining tables, the set operators present in the target language, and cursors as allowed in embedded SQL programs.

It is our belief that changing or extending some of the existing modules of System X to implement our algorithms will not pose serious problems.

APPENDIX A

The BNF Description of the System X Canonical Query Representation

Examples that show the complete tree structures of the CQR which conform to this grammar are given in Appendix E.

```
<s^> ::= (<type><quantifier>* <s>) | (<type><quantifier>* <np>)

<type> ::= Y/N | WH

<quantifier> ::= <quant> | variable | <np>

<quant> ::= ALL | WHICH | EXIST

<s> ::= (<ambig><v><np>* )

<np> ::= (<relation><coln><ambig><nom><features>)

<nom>         ::=         (<relation><coln><ambig><noun>)         |         (<relation>

          <coln><ambig><nom><s> | <relation> <coln><ambig><s^>)

<noun> ::= (<relation><coln><ambig><n-type><const> <index>)

<v> ::= (<ambig><v-type><verb>)

<ambig> ::= T | NIL

<const> ::= T | NIL

<relation> ::= relation name

<coln> ::= column name

<n-type> ::= COLV | NUMBER

<V-type> ::= RELATION | PREDICATE | COORDINATE | NEGATION | OPERATION

<index> ::= value | variable

<verb> ::= relation name | <operation> | <predicate> | NOT | OR | AND

<operation> ::= MAX | MIN | AVERAGE | TOTAL | QUANTITY

<predicate> ::= GREATERTHAN | LESSTHAN | GREATERTHANEQ | LESSTHANEQ | EQ |

          NOTEQ

<features> ::= a list of stored objects
```

APPENDIX B

The following is a BNF representation of the grammar that defines the logical form used by System X.

```
<query>::=<verification> | <retieval> | <aggregation>

<retrieval>::=<simplestmt> | <declaration>

<verification>::=(INT* <retrieval> | <agg_stmt>)

<aggregation>::=(SETX <variable><agg_calc>)

<simplestmnt>::=(RELATION <relname>(<attrname>*) (<constant>*)(<relop>))

<declaration>::=(<declop><variable><declaration> | <condition_clause>)

<agg-calc>::=(<aggop><variable><retrieval>)

<condition-clause>::<conjunction> | <rclause>

<conjunction>::=(AND<conjclause><conjclause>*)

<conjclause>::=<arithfunction> | <negation> | <selection> | <comparison> |
          <rclause> | <agg_calc>

<arithfunction>::=(<arithop><variable> <arithvalue><arithvalue>)

<comparison>::(<compop><variable><attrvalue>)

<negation>::=(NOT* <retrieval>)

<selection>::=(<selop><variable><retrieval>)

<rclause>::=(RELATION<relname>(<attrname>*) (<attrvalue>*)(<relop>*))

<relname>::=name of a base or virtual relation in the DB

<atrrname>::=DB attribute name

<declop>::=SETX | BAGX

<aggop>::=TOTAL | QUANTITY | AVERAGE

<arithop>::=PRODUCT|DIVISION|ADDITION|SUBTRACTION

<compop>::=GREATERTHAN|GREATERTHANEQ|LESSTHAN|
        LESSTHANEQ|NOTEQ|EQUAL
```

```
<selop>::=MAX|MIN

<relop>::==

<arithvalue>::=<number>|<variable>

<attrvalue>::=<variable>|<constant>
```

APPENDIX C

Our Addition to System X's LF is given below. Examples that show structures of LF that conform to a combination of this grammar and that given in Appendix B are presented in Appendix E.

```
<query>::=<verification> | <retrieval> | <aggregation> |<comparative>|<ExplicitStruct>
<comparative>::=(COND* <verification><retrieval> <retrieval>)
<ExplicitStruct>::=([INT*]        [NOT*][<ExplicitStruct>]        <declaration>*
        [<aggregatestmt>][<OrderByClause>][(<setcondition>)])
<setcondition>>::= SUPERSETOP | NOSETOP | NOTEMPTYSETOP | SUBSETOP
<OrderByClause>::=    (ORDERBY    <variable>    (SECONDARY    <variable>)
        <setcondition><declaration>)
<aggregatestmt>::= (aggop <variable>)
<numeralstmt> ::= (Numeral number)
```

APPENDIX D

Grammar for the Target Language

Examples showing the use of this target language are presented in Appendix E.

<statement>::=<allowed SQL stmt> | <assignment>|
    <whileloop>|<ifstmt>|<repeatloop>|<forloop>| <printstmt>|<procedurecall>

<asignment>::=<variable>":="<expression>

<whileloop>::="WHILE"<condition:expression> "DO" <body:statement>

<forloop>::= "FOR" <forvar:identifier> ":="<initialvalue:expression> "TO"
    <finalvalue:expression> "DO" <body:statement>

<repeatloop>::="REPEAT" <body:statementlist>"UNTIL"<condition:expression>

<ifstatement>::="IF"<predicate:expression> "THEN" <consequent:statement> ["ELSE"
    <alternate:statement>]

<printstmt>::="WRITE" "(" <writeparameters:writeparamaterlist> ")"

<procedurecall>::=<procedurename:identifier> ["(" <arguments:expressionlist> ")"]

<expression>::=<simple SQL stmt>|<variable>|<string>| <number>|<relation>

<relation>::=<operand1:simpleexpr><operator:relationop> <operand2:simpleexpr>

<variable>::=<identifier>|<indexedvar>

```
<indexedvar>::=<array:variable>"["<indeces:expressionlist>

<number>::=<integer>|<realnumber>

<operator>::= =|>|<|≥|≤|⊇ |⊆|≠ ∅
```

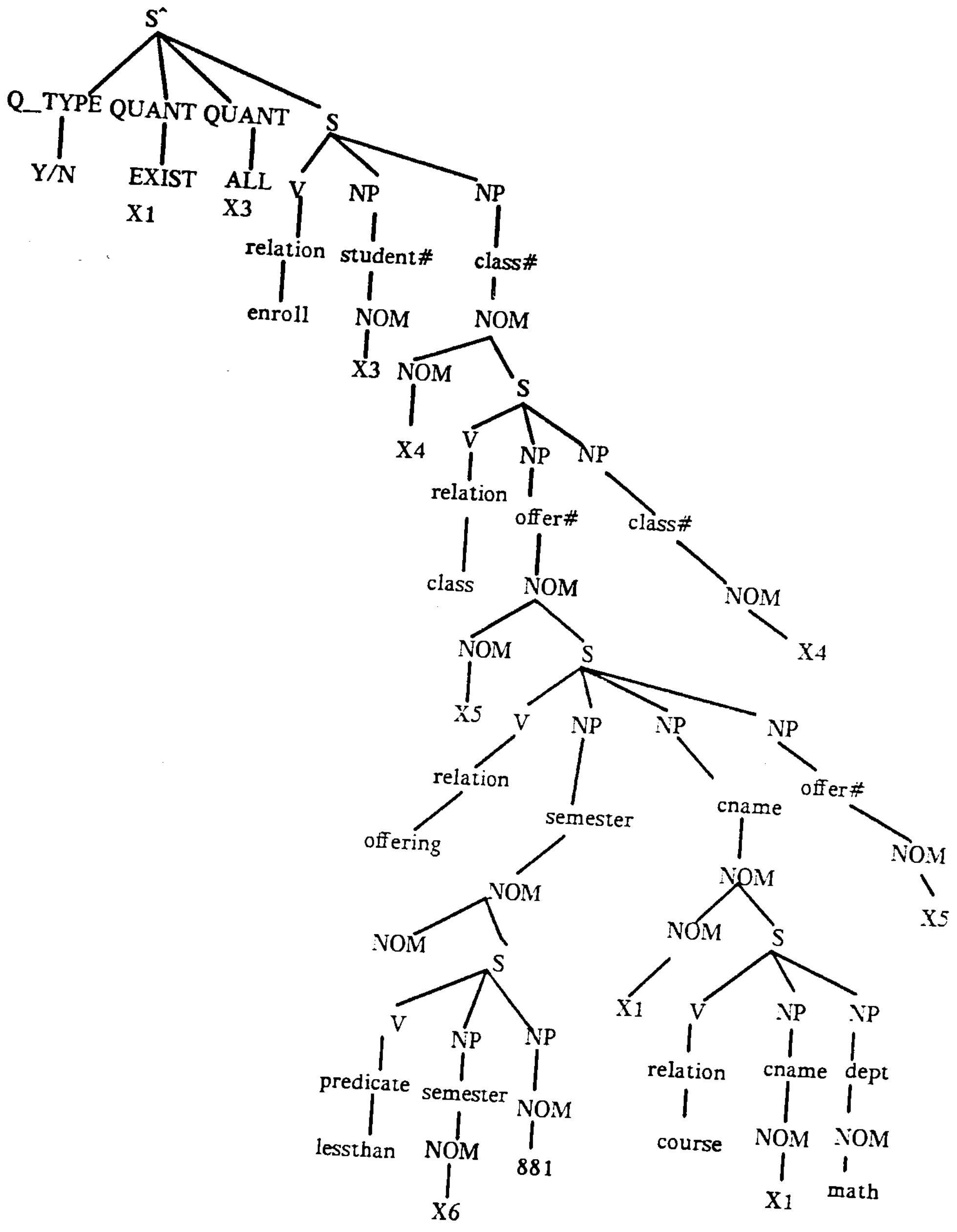
APPENDIX E

This Appendix presents some sample queries and shows how they are translated from CQR to LF and then from LF to our target language using the two algorithms presented in Chapter 6.

Example Query 1

Was a Math course taken by every student ?

The CQR for this query is the following. This example involves one EXIST and one ALL. The interpretation given to this query is "is there a particular math course which is taken by all students ?".



The LF for example query 1 is the following:

```
(INT*
 (SETX 'X3
  (RELATION ENROLL
   (STUDENT#)
   (X3)
   (=)))

 (ORDERBY 'X3
  (SECONDARY 'X1)
   (SUPERSETOP)
    (SETX 'X3
     (SETX 'X4
      (SETX 'X5
       (SETX 'X1
        (AND
         (RELATION ENROLL
          (STUDENT# CLASS#)
          (X3 X4)
          (= =))
         (RELATION CLASS
          (OFFER# CLASS#)
          (X5 X4)
          (= =))
         (RELATION OFFERING
          (SEMESTER CNAME OFFER#)
          (X6 X1 X5)
```

```
                        (= = =))
                     (LESSTHAN X6 '881)
                     (RELATION COURSE
                        (CNAME DEPT)
                        (X1 MATH)
                        (= =))))))))

(NOTEMPTYSETOP))
```

The target language for the example query 1 is as follows:

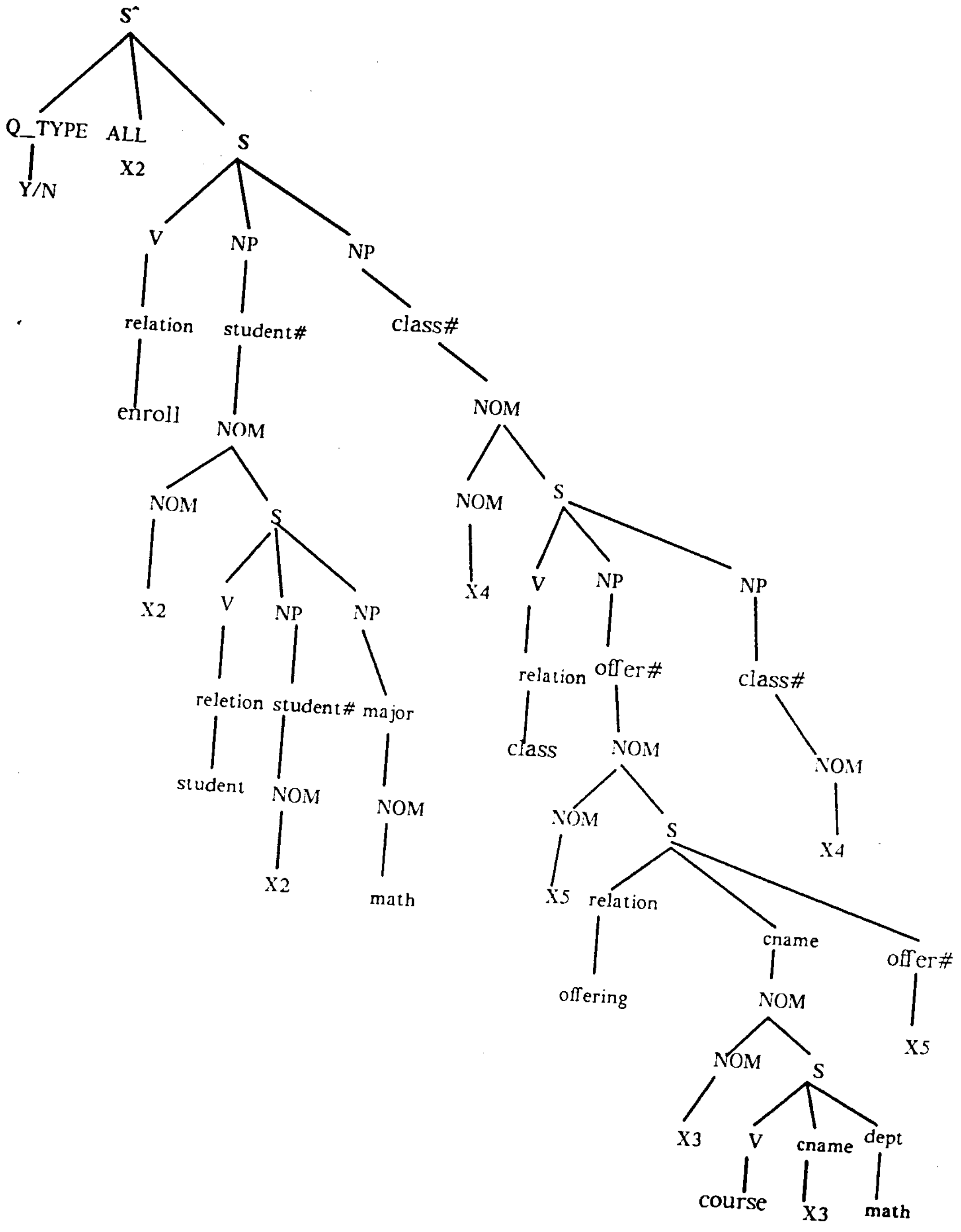
```
V1 :=
      SELECT UNIQUE A.STUDENT#
      FROM ENROLL A;

CREATE VIEW TEMP AS
      SELECT UNIQUE A.STUDENT#, D.CNAME
      FROM ENROLL A,CLASS B,OFFERING C,COURSE D
      WHERE A.CLASS#=B.CLASS#
      AND B.OFFER#=C.OFFER#
      AND C.SEMESTER<'881'
      AND C.CNAME=D.CNAME
      AND D.DEPT='MATH';
TEMPVAR :=
      SELECT UNIQUE CNAME
      FROM TEMP;
FOR I := 1 to (Number of elements in TEMPVAR)
   DO;
```

```
        SETi :=
            SELECT STUDENT
            FROM TEMP
            WHERE CNAME = TEMPVAR i;
        IF SETi ⊇ V1
            then add CNAME i to V2
END;

IF V2 ≠ ∅
    THEN write ("yes");
    ELSE write ("no");
```

Example Query 2

Does every math major take a math course ?

The CQR for this query is as follows. This is an example of a query with only one ALL. The interpretation given to this query is "is it the case that every math major takes at least one math course?".



The LF for Example query 2 is as follows:

```
(INT*
    (SETX 'X2
        (RELATION STUDENT
            (STUDENT# MAJOR)
            (X2 'MATH)
            (= =)))
    (SETX 'X2
        (SETX 'X4
            (SETX 'X5
                (SETX 'X3
                    (AND
                        (RELATION ENROLL
                            (STUDENT# CLASS#)
                            (X2 X4)
                            (= =))
                        (RELATION STUDENT
                            (STUDENT# MAJOR)
                            (X2 MATH)
                            (= =))
                        (RELATION CLASS
                            (OFFER# CLASS#)
                            (X5 X4)
                            (= =))
                        (RELATION OFFERING
                            (CNAME OFFER#)
                    (X3 X5)
                    (= =))
                 (RELATION COURSE
                    (CNAME DEPT)
                    (X3 MATH)
                    (= =)))))))

      (SUPERSETOP))
```

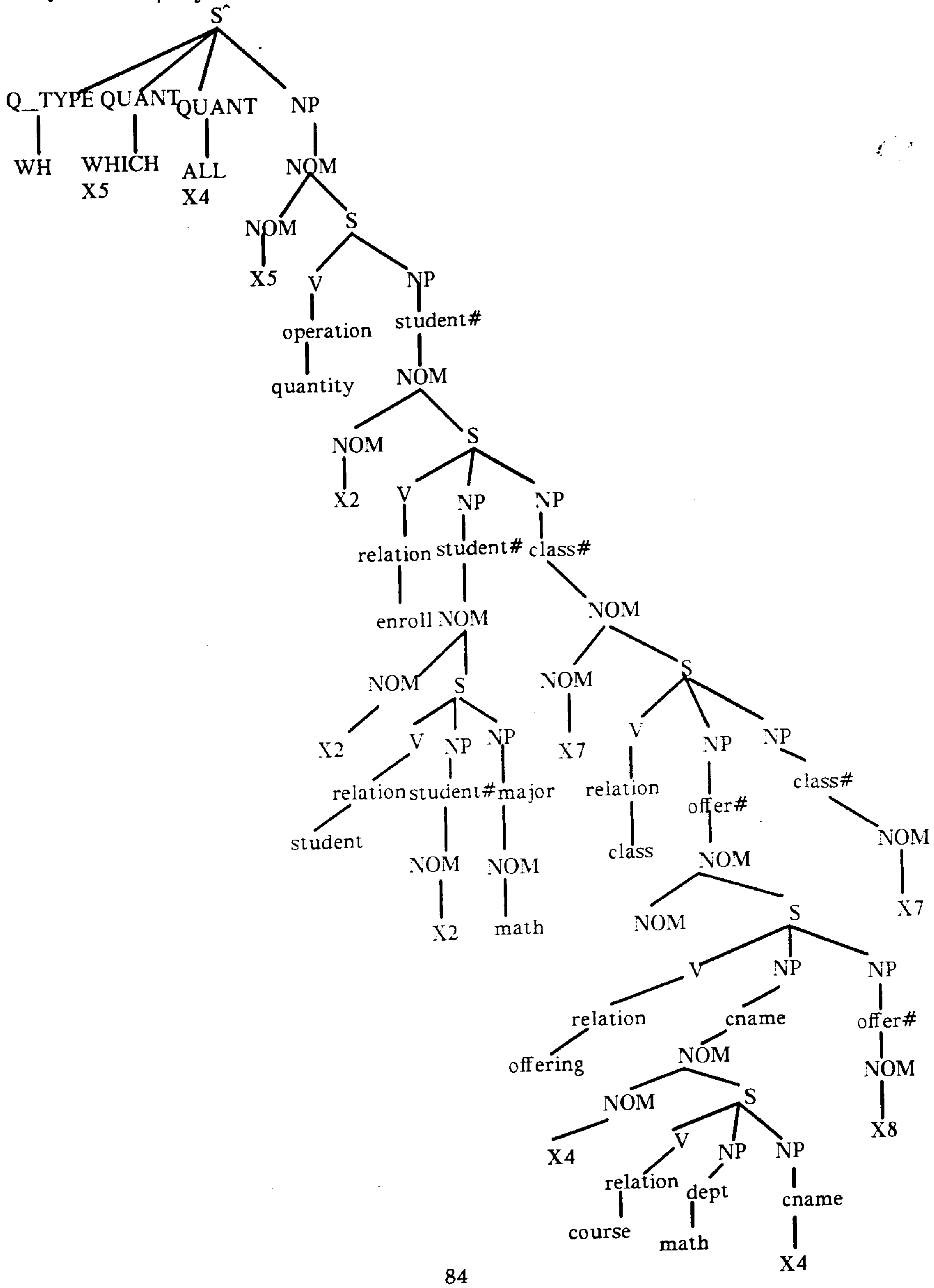
The target language for example query 2 is given below:

```
      V1 :=
            SELECT A.STUDENT#
            FROM STUDENT A
            WHERE A.MAJOR='MATH';
      V2 :=
            SELECT A.STUDENT#
            FROM ENROLL A,STUDENT B,CLASS C, OFFERING D,COURSE E
            WHERE A.STUDENT#=B.STUDENT#
            AND A.CLASS#=C.CLASS#
            AND B.MAJOR='MATH'
            AND C.OFFER#=D.OFFER#
            AND D.CNAME=E.CNAME
            AND E.DEPT='MATH';

      IF V2 ⊇ V1
         THEN write ("yes");
         ELSE write ("no");
```

Example Query 3

How many math majors take every math course ?

The CQR for this query is as follows:



The LF for Example query 3 is given below:

```
(
  (SETX 'X4
    (RELATION COURSE
      (CNAME DEPT)
      (X4 'MATH)
      (= =)))

  (TOTAL 'X5)
      (ORDERBY 'X2
        (SECONDARY 'X4)
          (SUPERSETOP)
            (SETX 'X2
              (SETX 'X7
                (SETX 'X8
                  (SETX 'X4
                    (AND
                      (RELATION ENROLL
                        (STUDENT# CLASS#)
                        (X2 X7)
                        (= =))
                      (RELATION STUDENT
                        (STUDENT# MAJOR)
                        (X2 MATH)
                        (= =))
                      (RELATION CLASS
                        (OFFER# CLASS#)
                        (X8 X7)
                        (= =))
                    (RELATION OFFERING
                        (CNAME OFFER#)
                        (X4 X8)
                        (= =))
                    (RELATION COURSE
                        (DEPT CNAME)
                        ('MATH X4)
                        (= =)))))))))

(NOSETOP))
```

The target language for example query 3 is as given below:

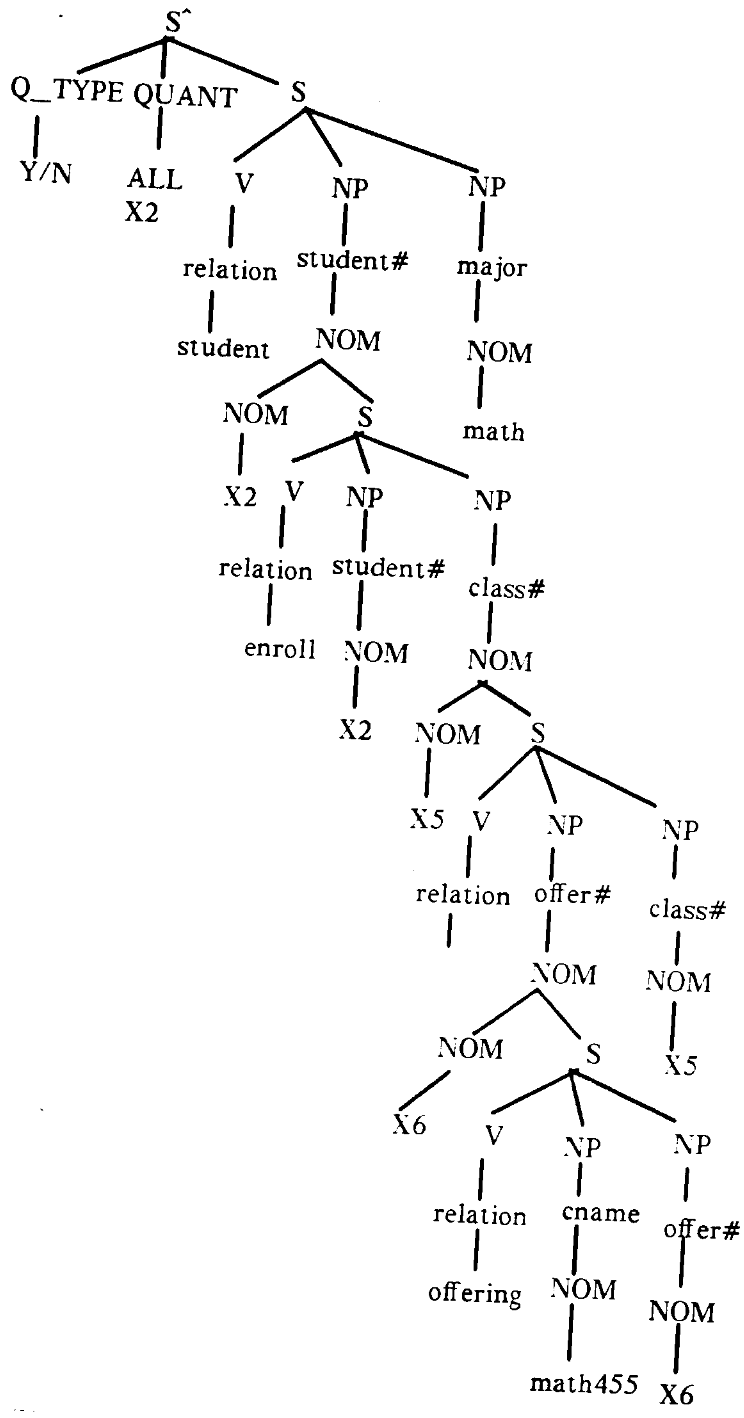
```
V1 :=
    SELECT UNIQUE A.CNAME
    FROM COURSE A
    WHERE A.DEPT='MATH';

CREATE VIEW TEMP AS
    SELECT D.CNAME, A.STUDENT#
    FROM ENROLL A,STUDENT B,CLASS C, OFFERING D, COURSE E
    WHERE A.STUDENT#=B.STUDENT#
    AND A.CLASS#=C.CLASS#
    AND B.MAJOR='MATH'
    AND C.OFFER#=D.OFFER#
    AND D.CNAME = E.CNAME
```

```
        AND E.DEPT = 'MATH';
TEMPVAR :=
        SELECT UNIQUE STUDENT#
        FROM TEMP;

FOR I := 1 to (Number of elements in TEMPVAR)
    DO;
        SETi :=
            SELECT CNAME
            FROM TEMP
            WHERE STUDENT# = TEMPVAR i;
        IF SETi ⊇ V1
            then add STUDENT# i to V2;
END;

WRITE Count of (V2);
    {The procedure Operator-Evaluate gives this count}
```

Example Query 4

Do only math majors take math455 ?

The CQR for this query is given below. This is a case of a query that has only represented as an ALL at the quant level of the CQR.



The LF for Example Query 4 is given below.

```
(INT*
    (SETX 'X2
        (SETX 'X5
            (SETX 'X6
                (AND
                    (RELATION ENROLL
                        (STUDENT# CLASS#)
                        (X2 X5)
                        (= =))
                    (RELATION CLASS
                        (OFFER# CLASS#)
                        (X6 X5)
                        (= =))
                    (RELATION OFFERING
                        (CNAME OFFER#)
                        (MATH455 X6)
                        (= =))))))

    (SETX 'X2
        (SETX 'X5
            (SETX 'X6
                (AND
                    (RELATION STUDENT
                        (STUDENT# MAJOR)
                        (X2 'MATH)
                        (= =))
        (RELATION ENROLL
            (STUDENT# CLASS#)
            (X2 X5)
            (= =))
        (RELATION CLASS
            (OFFER# CLASS#)
            (X6 X5)
            (= =))
        (RELATION OFFERING
            (CNAME OFFER#)
            (MATH455 X6)
            (= =))))))

(SUPERSETOP))
```

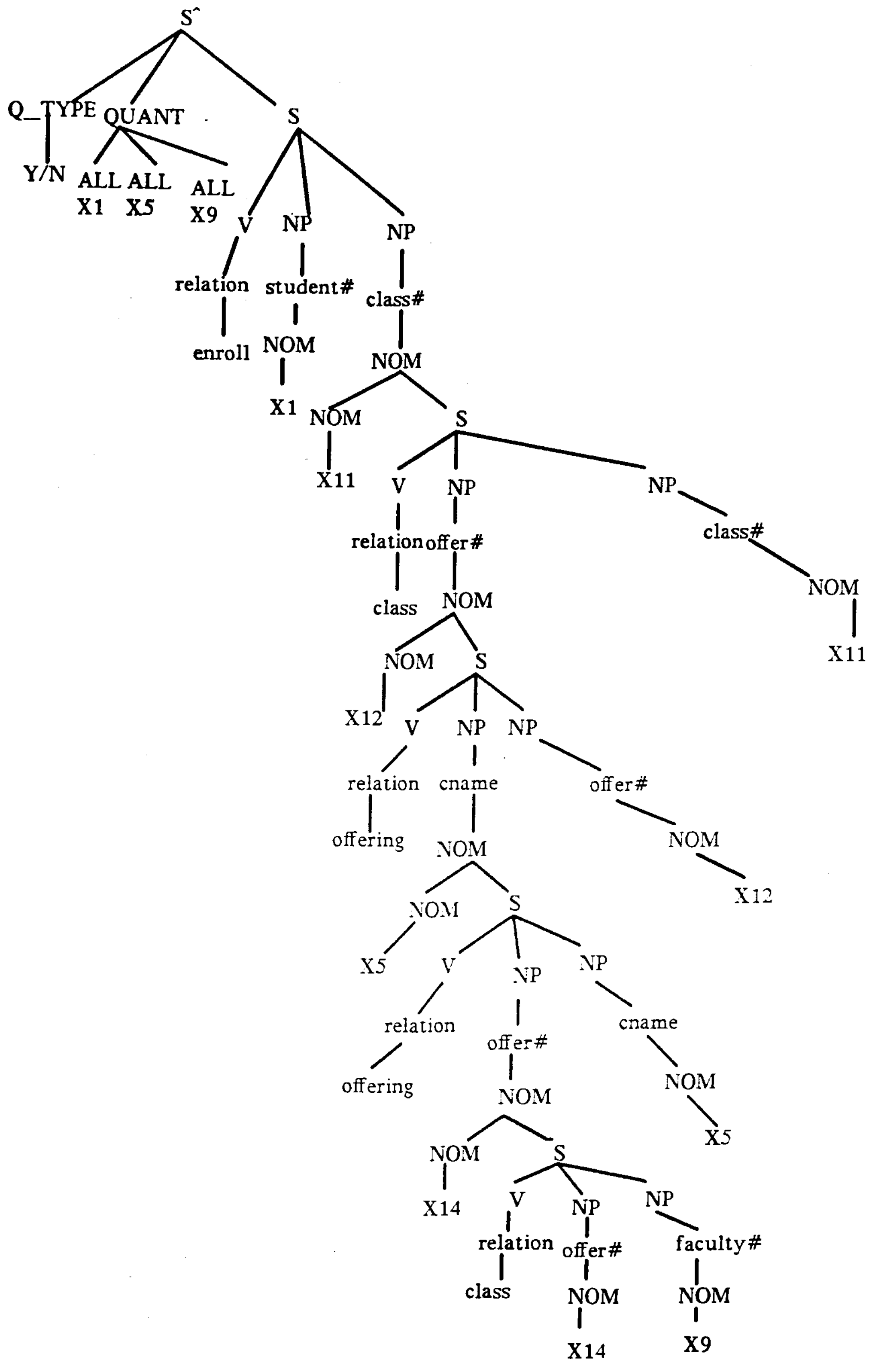
The target language for example query 4 is given below:

```
V1 :=
        SELECT UNIQUE A.STUDENT#
        FROM ENROLL A,CLASS B, OFFERING C
        WHERE A.CLASS#=B.CLASS#
        AND B.OFFER#=C.OFFER#
        AND C.CNAME='MATH455' ;

V2 :=
        SELECT UNIQUE A.STUDENT#
        FROM STUDENT A, ENROLL B, CLASS C, OFFERING D
        WHERE A.STUDENT# = B.STUDENT#
        AND A.MAJOR = 'MATH'
        AND B.CLASS# = C.CLASS#
        AND C.OFFER# = D.OFFER#
        AND D.CNAME = 'MATH455';

IF V2 $\supseteq$ V1
    THEN WRITE ("yes");
    ELSE WRITE ("no");
```

Example Query 5

Do all students take every course from every instructor ?

The CQR for this query is given below. This is a case of three ALL's at the quant level of the CQR.



The LF for Example Query 5 is given below:

```
(INT*
    (SETX 'X1
        (RELATION ENROLL
            (STUDENT#)
            (X1)
            (=)))

    (SETX 'X5
        (SETX 'X14
            (SETX 'X9
                (AND
                    (RELATION OFFERING
                        (OFFER# CNAME)
                        (X14 X5)
                        (= =))
                    (RELATION CLASS
                        (OFFER# FACULTY#)
                        (X14 X9)
                        (= =))))))

    (ORDERBY X1
        (SECONDARY X5)
            (SUPERSETOP)
                (SETX 'X5
                    (SETX 'X1
                        (SETX 'X11
                    (SETX 'X12
                       (AND
                          (RELATION ENROLL
                             (STUDENT# CLASS#)
                             (X1 X11)
                             (= =))
                          (RELATION CLASS
                             (OFFER# CLASS#)
                             (X12 X11)
                             (= =))
                          (RELATION OFFERING
                             (CNAME OFFER#)
                             (X5 X12)
                             (= =)))))))

(SUPERSETOP))
```

The target language for example query 5 is given below:

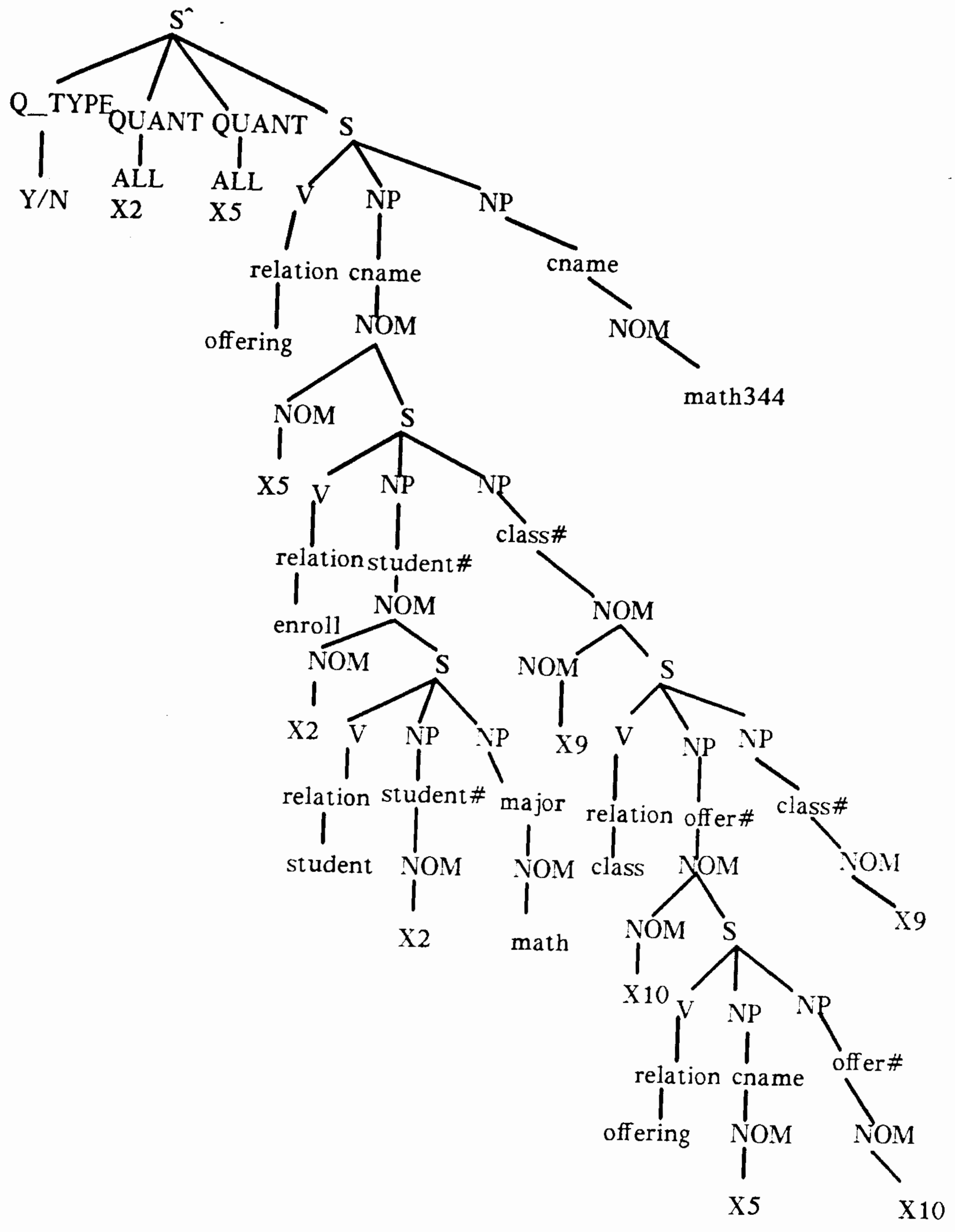
```
V1 :=
      SELECT UNIQUE A.STUDENT#
      FROM ENROLL A;

V2 :=
      SELECT UNIQUE A.CNAME
      FROM OFFERING A, CLASS B
      WHERE A.OFFER# = B.OFFER# ;

CREATE VIEW TEMP AS
```

```
        SELECT UNIQUE C.CNAME, A.STUDENT#
        FROM ENROLL A, CLASS B, OFFERING C
        WHERE A.CLASS# = B.CLASS#
        AND B.OFFER# = C.OFFER#

TEMPVAR :=
        SELECT UNIQUE STUDENT#
        FROM TEMP;
FOR I := 1 to (Number of elements in TEMPVAR)
    DO;
        SETi :=
            SELECT CNAME
            FROM TEMP
            WHERE STUDENT# = TEMPVAR i;
        IF SETi $\supseteq$ V2
            THEN add STUDENT# i to V3;
END;
IF V3 $\supseteq$ V1
    THEN WRITE ("yes")
    ELSE WRITE ("no");
```

Example Query 6

Do all math majors take only math344 ?

The CQR for this query is given below:



The LF for Example Query 6 is given below:

```
(INT*
   (SETX 'X2
      (RELATION STUDENT
         (STUDENT# MAJOR)
         (X2 'MATH)
         (= =)))
   (SETX 'X5
      (RELATION OFFERING
         (CNAME CNAME)
         (X5 'MATH344)
         (= =)))
   (ORDERBY 'X2
      (SECONDARY 'X5)
         (SUBSETOP)
            (SETX 'X5
               (SETX 'X2
                  (SETX 'X9
                     (SETX 'X10
                        (AND
                           (RELATION OFFERING
                              (CNAME )
                              (X5 )
                              (= ))
                           (RELATION ENROLL
                              (STUDENT# CLASS#)
```

```
                (X2 X9)
                (= =))
            (RELATION CLASS
                (OFFER# CLASS#)
                (X10 X9)
                (= =))
            (RELATION OFFERING
                (CNAME OFFER#)
                (X5 X10)
                (= =))))))
(SUPERSETOP))
```

The target language for example query 6 is given below:

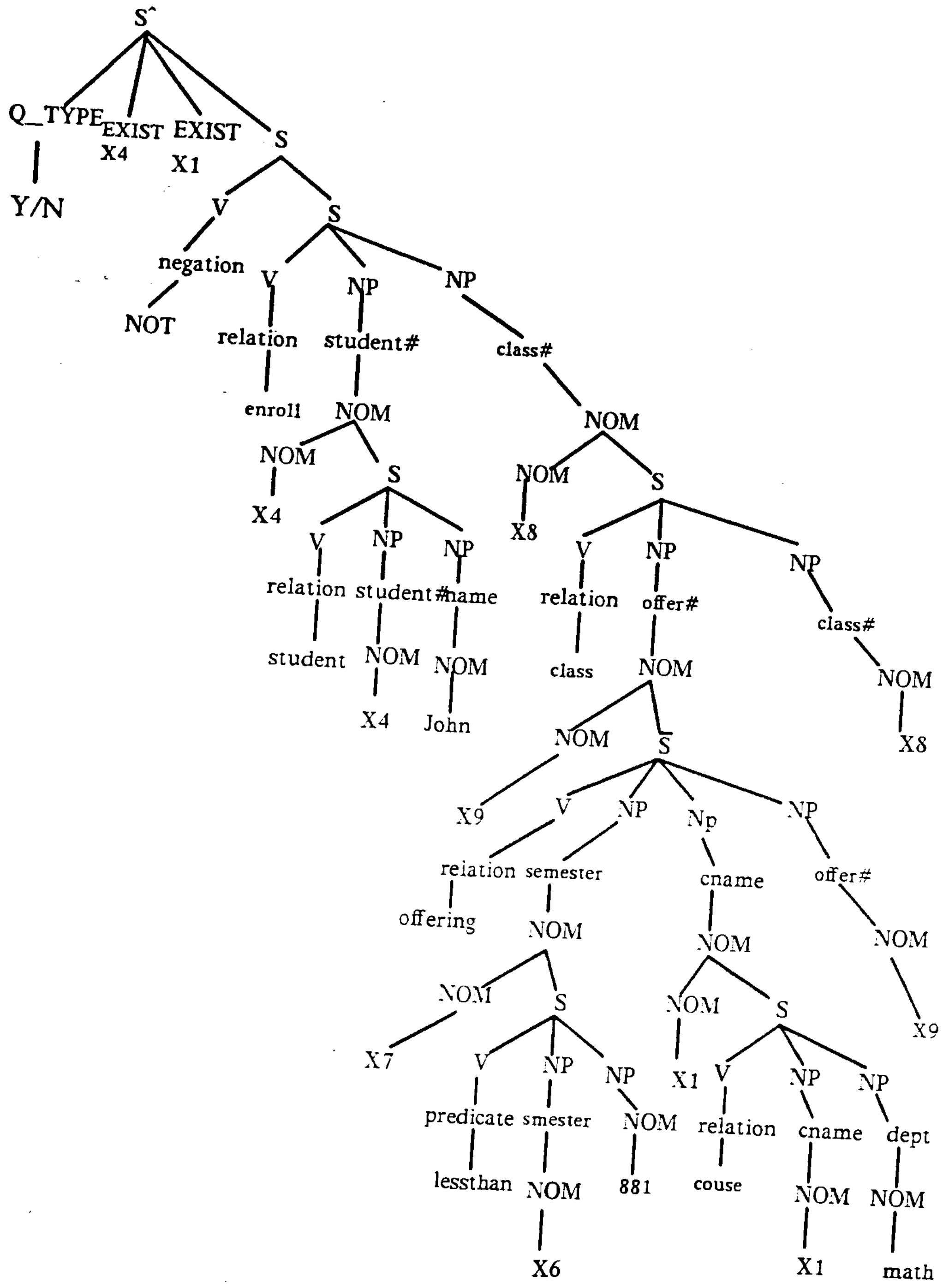
```
V1 :=
    SELECT UNIQUE A.STUDENT#
    FROM STUDENT A
    WHERE A.MAJOR = 'MATH';

V2 :=
    SELECT UNIQUE A.CNAME
    FROM OFFERING A
    WHERE A.CNAME = 'MATH344';
CREATE VIEW TEMP AS
    SELECT A.CNAME, C.STUDENT#
    FROM OFFERING A, ENROLL B,STUDENT C, CLASS D, OFFERING E
    WHERE A.CNAME = E.CNAME
    AND B.STUDENT# = C.STUDENT#
```

```
        AND C.MAJOR = 'MATH' AND D.CLASS# = B.CLASS#
        AND D.OFFER# = E.OFFER#;
TEMPVAR :=
        SELECT UNIQUE STUDENT#
        FROM TEMP;
FOR I := 1 to (Number of elements in TEMPVAR)
    DO;
        SETi := SELECT CNAME
        FROM TEMP
        WHERE STUDENT# = TEMPVAR i ;
        IF SETi ⊆ V2
            THEN add STUDENT# i to V3
END;
IF V3 ⊇ V1
    THEN WRITE ("yes");
    ELSE WRITE ("no");
```

Example Query 7

Did John not take every math course ?

The CQR for this query is given below. This is a case of a query involving universal quantifier and negation which gets interpreted as EXIST's at the CQR level. If John is found to have taken all math courses, the algorithm answers "no", otherwise it answers "yes".



The LF for Example Query 7 is given below:

```
(INT*
   (SETX 'X5
      (RELATION STUDENT
         (STUDENT# NAME)
         (X5 'JOHN)
         (= =)))

   (SETX 'X1
      (RELATION COURSE
         (DEPT CNAME)
         ('MATH X1)
         (= =)))

   (NOT*
      (ORDERBY 'X5
         (SECONDARY 'X1)
            (SUPERSETOP)
               (SETX 'X1
                  (SETX 'X5
                     (SETX 'X8
                        (SETX 'X9
                           (SETX 'X4
                              (AND
                                 (RELATION ENROLL
                                    (STUDENT# CLASS#)
                                    (X5 X8)
```

```
                                  (= =))
                                (RELATION STUDENT
                                  (STUDENT# NAME)
                                  (X5 'JOHN)
                                  (= =))
                                (RELATION CLASS
                                  (CLASS# OFFER#)
                                  (X8 X9)
                                  (= =))
                                (RELATION OFFERING
                                  (OFFER# CNAME SEMESTER)
                                  (X9 X1 X4)
                                  (= = =)))))))))))
      (SUPERSETOP))
```

The target language for example query 7 is given below:

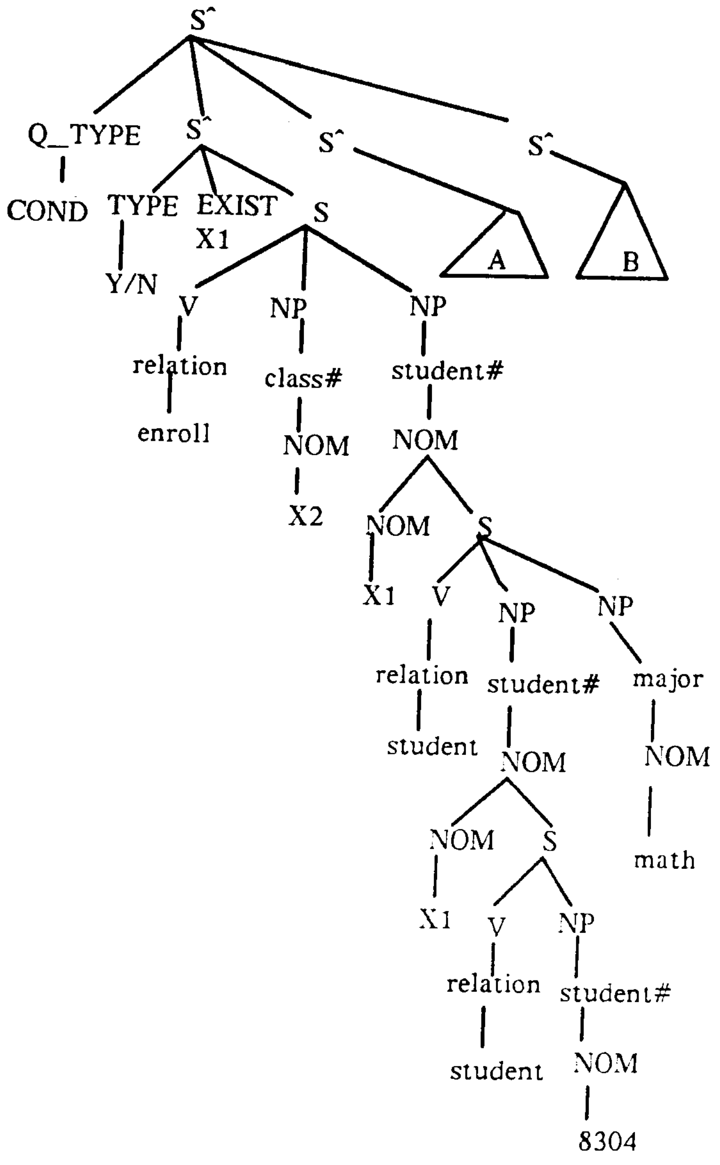
```
      V1 :=
          SELECT A.STUDENT#
          FROM STUDENT A
          WHERE A.NAME = 'JOHN';

      V2 :=
          SELECT A.CNAME
          FROM COURSE A
          WHERE A.DEPT = 'MATH';
      CREATE VIEW TEMP AS
          SELECT D.CNAME,B.STUDENT#
```

```
    FROM ENROLL A, STUDENT B, CLASS C, OFFERING D, COURSE E
    WHERE A.STUDENT# = B.STUDENT#
    AND B.NAME = 'JOHN'
    AND A.CLASS# = C.CLASS#
    AND C.OFFER# = D.OFFER#
    AND D.CNAME =E.CNAME
    AND D.SEMESTER < '881'
    AND E.DEPT = 'MATH';
TEMPVAR :=
    SELECT UNIQUE STUDENT#
    FROM TEMP;
FOR I := 1 to (Number of elements in TEMPVAR)
  DO;
    SETi :=
      SELECT CNAME
      FROM TEMP
      WHERE STUDENT# = TEMPVAR i;
    IF SETi $\supseteq$ V2
      THEN add STUDENT# i to V3;
END;
IF V3 $\supseteq$ V1
  THEN WRITE ("no");
  ELSE WRITE ("yes");
```

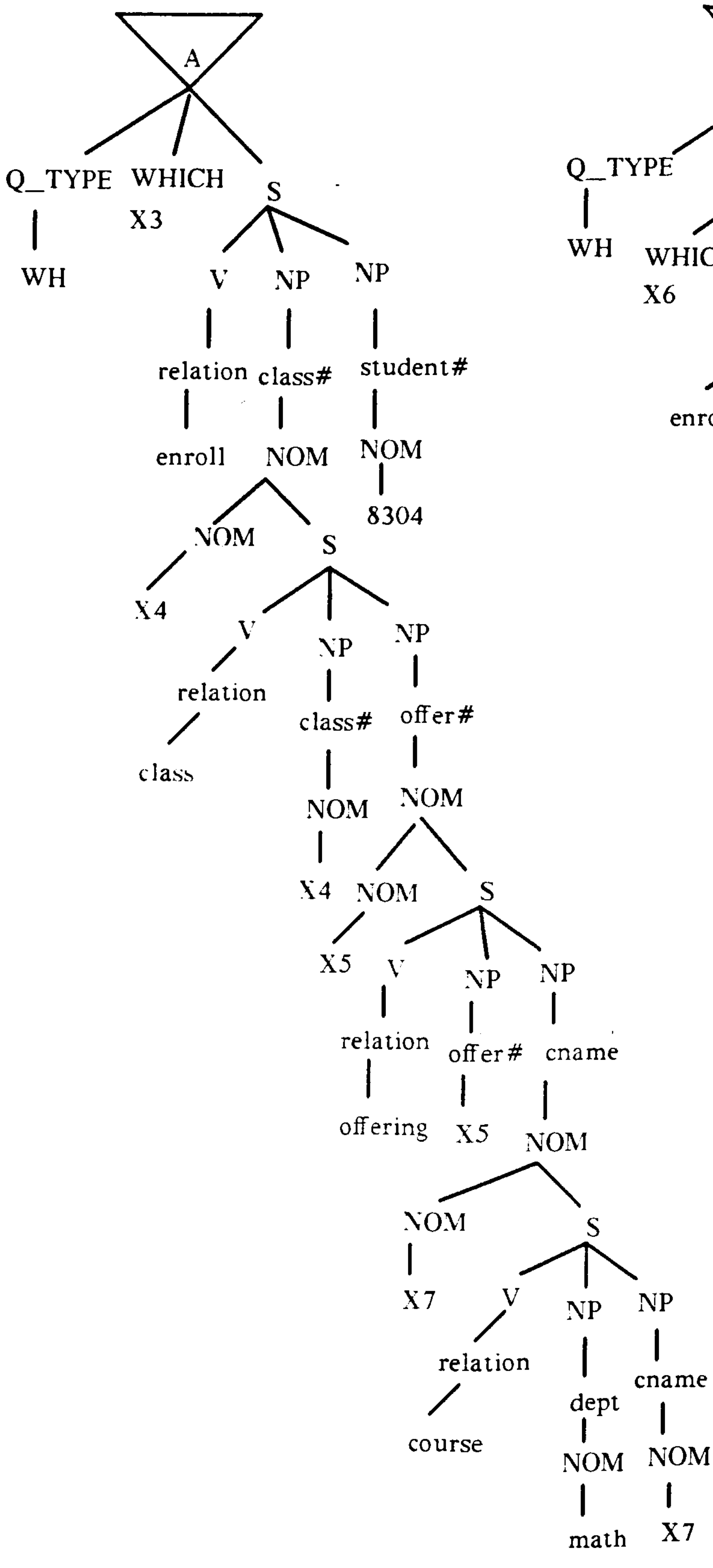
Example Query 8

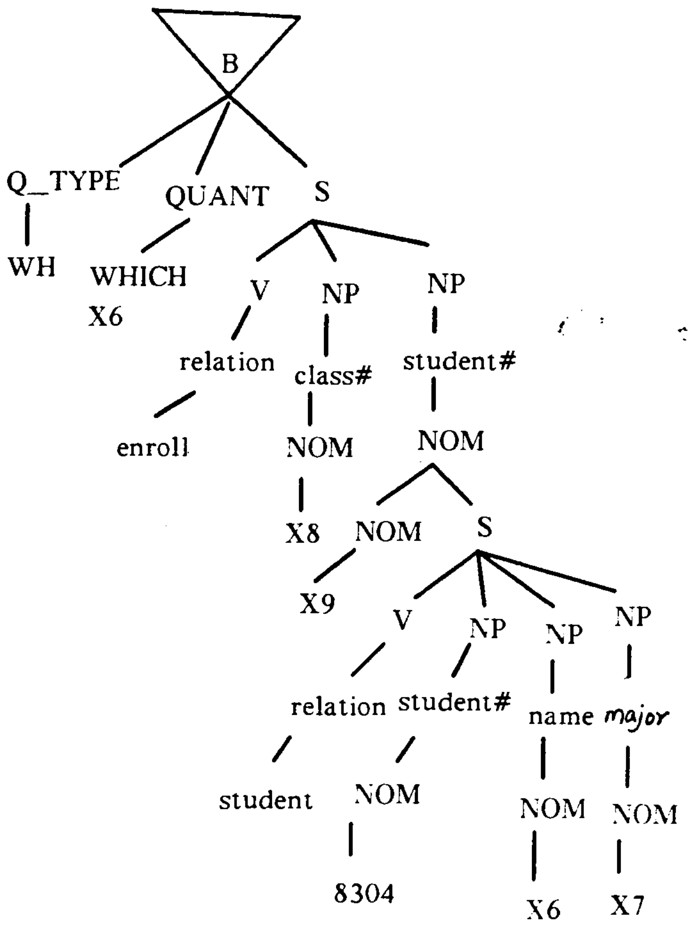
If student number 8304 is a math major, give me the math courses he is taking else give me his name and dept.

This is a case of a query which is of type COND; and the CQR for this query is given below.



If Student Number 8304 is a math Major, Give me the math Courses he Takes else Give me his Name and Dept


A
Q_TYPE
WH
WHICH
X3
S
V
relation
enroll
NP
class#
NOM
NOM
X4
S
V
relation
class
NP
class#
NOM
X4
NP
offer#
NOM
NOM
X5
S
V
relation
offering
NP
offer#
X5
NP
cname
NOM
NOM
X7
S
V
relation
course
NP
dept
NOM
math
NP
cname
NOM
X7
NP
student#
NOM
8304


B
Q_TYPE
WH
QUANT
WHICH
X6
S
V
relation
enroll
NP
class#
NOM
X8
NP
student#
NOM
NOM
X9
S
V
relation
student
NP
student#
NOM
8304
NP
name
NOM
X6
NP
major
NOM
X7

The LF for Example Query 8 is given below:

```
(COND*
   (INT*
      (SETX 'X1
         (SETX 'X2
            (AND
               (RELATION ENROLL
                  (CLASS# STUDENT#)
                  (X2 X1)
                  (= =))
               (RELATION STUDENT
                  (STUDENT# MAJOR)
                  (X1 'MATH)
                  (= =))
               (RELATION STUDENT
                  (STUDENT#)
                  ('8304)
                  (=)))))))

   (SETX 'X3
      (SETX 'X4
         (SETX 'X5
            (AND
               (RELATION ENROLL
                  (CLASS# STUDENT#)
                  (X4 '8304)
```

```
                    (= =))
                  (RELATION CLASS
                    (CLASS# OFFER#)
                    (X4 X5)
                    (= =))
                  (RELATION OFFERING
                    (OFFER# CNAME DEPT)
                    (X5 X3 'MATH)
                    (= = =)))))))

(SETX 'X6
  (SETX 'X8
    (SETX 'X9
      (SETX 'X6
        (SETX 'X7
          (AND
            (RELATION ENROLL
              (CLASS# STUDENT#)
              (X8 X9)
              (= =))
            (RELATION STUDENT
              (STUDENT# NAME MAJOR)
              (X9 X6 X7)
              (= = =))
            (RELATION STUDENT
              (STUDENT#)
              (8304)
```

(=))))))))

The target language for the Example query 8 is given below:

```
IF
    (POS
    SELECT STUDENT#
    FROM ENROLL A,STUDENT B
    WHERE A.STUDENT# = B.STUDENT#
    AND B.MAJOR = 'MATH
    AND B.STUDENT# = '8304;)

THEN
    (SELECT C.CNAME
    FROM ENROLL A, CLASS B, OFFERING C
    WHERE A.CLASS# = B.CLASS#
    AND A.STUDENT# = '8304
    AND B.OFFER# = C.OFFER#
    AND C.DEPT = 'MATH ;)
ELSE
(SELECT B.NAME,B.MAJOR
FROM ENROLL A,STUDENT B
WHERE A.STUDENT# = B.STUDENT#
AND B.STUDENT# = '8304 ;)
```

REFERENCES

[BLT 84] Ballard, B.W.,Lusth, J.C., Tinkham.N.L., "LDC_1 : A Transportable Knowledge Based Natural Language Processor for Office Environments", *ACM Transactions on Office Information Systems,* Vol. 2, No. 1, Jan. 1984, pg. 1-25.

[BoFr 85] Thompson, B.H., and Thompson, F.B., "ASK is Transportanle in Half a Dozen Ways", *ACM Transactions on Office information Systems,* Vol. 3, No. 2, April 1985, pg. 185-203.

[Codd 70] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM,* 1970, 13(6), pg. 377-387.

[Codd 79] Codd, E. F., "Extending the Database Relational Model to Capture more Meaning", *ACM Transactions on Database Systems,* 1979, 4(4) pg. 397-434.

[Damerau 81] Damerau, F.J., "Operating Statistics for the Transformational Question Answering System"; *American Journal of Computational Linguistics*, Vol. 7, No. 1, Jan-Mar 1981.

[Damerau 85] Damerau, F.J., "Problems and Some Solutions in Customization of Natural Language Database Front Ends", *ACM Transactions on Office Information Systems*, Vol. 3, No. 2, 1985, pg. 165-184.

[Date 86a] Date, C.J., *An Introduction to Database Systems*, Fourth edition. Reading, Mass: Addison-Wesley, 1986.

[Date 86b] Date, C.J., "A Critique of the SQL Database Language",*Relational Database, Selected Writings*, Addison-Wesley Publishing Company, 1986.

[Hall 87] Hall, G., "Report on LF to SQL Translation", *Technical Report,LCCR,school of Computing Science,* Simon Fraser University, 1987.

[JKVSTW 85] Jarke, M., Krause, J., Vassillion, Y., Stohr, E., Turner, J., and White, N. "Evaluation and Assessment of a Domain_independent Natural Language Query System", *Database Engineering,* Sept 1985 Vol. 8, No. 3.

[Johnson 84] Johnson, D.E., "Design of a Robust, Portable Natural Language Interface Grammar", *Technical Report RC 10867, IBM Thomas J. Watson Research Laboratory,* 1984.

[LOZ 85] Lehmann, H., Ott, N., Zoepptritz, M., "A Multilingual Interface to Database"; *Database Engineering,* September 1985, Vol. 8, No. 3, 1985.

[LukKl 85] Luk, W.S., and Kloster, S., " English Language from SQL", *Technical Report,* LCCR, school of Computing Science, Simon Fraser University, #85-8, 1985.

[MAGP 85] Martin, P., Appelt, D.E., Grosz, B.J., and Pereira, F., "TEAM :An Experimental Transportable Natural Language Interface" *Database Engineering,* September 1985, Vol. 8, No. 3.

[McFet 87] McFetridge, P., "BNF Description of the Canonical Query Representation", *Technical Report,* Laboratory for Computers and Communications Research, school of Computing Science, SFU, 1987.

[MHCL 87] MCFetridge, P., Hall, G., Cercone, N., and Luk, W.S., "System X: a portable Natural Language Interface", *Technical Report,* Laboratory for Computer and Communications Research (LCCR), school of Computing Science, Simon Fraser University 1987.

[Petrick 84] Petrick, S. R., "Natural Language Database Query Systems", *Technical Report* RC 10508, IBM Thomas J. Watson Research Laboratory, 1984.

[Pylyshyn 85] Pylyshyn, Z.W., "Alternatives to the use of Natural Language in interfacing to Databases", *Database Engineering,* Sept 1985, Vol. 8, No. 3.

[PyKi 85] Pylyshyn, Z.W., Kittridge, R.I., "Databases and Natural Language Processing", *Database Engineering,* September 1985, Vol. 8 No. 3.

[SWP 82] Schneider, G. M., Weingart, S. W., Perlman, D. M., *An Introduction to Programming and Problem Solving with Pascal,* 2nd ed., John Wiley & Sons 1982.

[SlJu 85] Slocum, J., and Justus, C.F. "Transportability to other Languages: The Natural Language Processing Project in the AI Program at MCC", *ACM Transactions on Ofice Information Systems,* Vol. 3, No. 2, April 1985, pg 204-230.

[Stone 88] Stonebraker, M., "Future Trends in Data Base Systems", *Proceedings of Fourth International Conference on Data Engineering,* Feb 1-5, 1988, pg. 222-231.

[SWKH 76] Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES", *ACM Transaction on Database Systems,* 1(3), Sept, 1976.

[Ullman 82] Ullman, Jeffrey D., *Principles of Database Systems,* 2nd ed., Computer Science Press, Rockville, Maryland, 1982.

[Zloff 77] Zloff M. M., "Query-By-Example: A Database Language", *IBM System Journal* 16(4): 324-343, 1977.