# FRONTEND FOR CYRANO META MODEL

by

Nikhil Samant

Report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF INFORMATION SYSTEMS

APPROVED:

[illegible]

Dr. Csaba Egyhazy, Chair

[illegible]

Dr. William Frakes

[illegible]

Dr. Emile Haddad

December 2, 1994

Falls Church, Virginia

c.2

LD
5655
V851
1994
S263
C.2

# FRONTEND FOR CYRANO META MODEL

by

Nikhil Samant

Committee Chair: Dr. Csaba Egyhazy

Department of Computer Science

## (ABSTRACT)

A frontend for new meta model Cyrano was designed and developed. Frontend development concentrated on the query language specification and the translation of SQL query to Cyrano representation.

The data models considered in the design included two relational, one object oriented, one hierarchical and one network model. A sample SQL query was translated into derived Cyrano classes to resolve the heterogeneity in database names and attribute names.

The prototype implementation included a lexical analyzer which recognized the SQL tokens, a parser which validated the SQL grammar and a translator which translated a SQL query into an equivalent Cyrano representation.

## ACKNOWLEDGEMENT

A sincere thanks to Dr. Leonard Gallagher from National Institute of Standards and Technology who send me his two papers on SQL3 for CALS application. These two papers guided initial search for SQL3 documents.

I am also grateful to Donald Deutsch from Sybase corporation for allowing me to electronically download the drafts for the SQL2 and the SQL3 standard.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1.
# INTRODUCTION

A database system consists of a database management system (DBMS) and databases built according to the DBMS's data model. A Federated database system is defined as a collection of cooperating but autonomous component database systems. In federated database systems, the component database management systems are integrated to various degrees. The federated database management system (FDBMS) integrates the component databases by controlling and coordinating the manipulation of these components.

A meta model for FDBMS is an integrating model that is capable of embracing the essence of all underlying data models in the federation. Cyrano is a meta model for federated database systems based on ideas from rule based and object oriented models. Cyrano sees the world full of objects in external databases and groups them into classes of similar objects. Cyrano resolves database heterogeneity by means of derived classes.

Roxanne is the core implementation of the Cyrano meta model which resolves the heterogeneity. Roxanne can be considered as backend of Cyrano which processes the actual query. Roxanne is being developed by another student working on his PhD. dissertation.

The purpose of this project was to suggest a suitable frontend for the Cyrano meta model. A good frontend user interface with flexible query language, is a necessary component of any database management system. This project primarily concentrates on the data access component of frontend which involves mapping of the SQL queries into appropriate Cyrano classes. The SQL is relational data language and Cyrano is object oriented and rule based model, so a smooth translation from relational to object oriented and rule based model is a key problem area of this project. This project also makes an attempt to show that heterogeneity in names and attributes can be resolved more efficienty at the fronend than at the backend.

This report describes the design and implementation of the frontend to the Cyrano meta model. The method adopted to translate the query into Cyrano classes, the rationale behind the translation and the problems encountered are discussed in detail. Due to space constraint only the relevant features of the Cyrano meta model are described here. The code for the prototype is listed in the appendix A and is documented using the guidelines suggested in [FRAKES].

# CHAPTER 2.
# CYRANO META MODEL

The various aspects of the Cyrano meta model that are relevant to the design and development of frontend are described here.

## 2.1. Federated Database Systems and Meta Model.

A federated database system is a collection of different database systems. A federated system may have various databases systems with different database models, languages, and services. Each database system in a federation can have a different data model. A meta model is defined as a model behind the existing data models.

A meta model must be capable of encompassing the essence of all underlying data models in the federation. Thus, it should be able to provide a mechanism for the temporary storage of incompatible data and its associated semantics. Once these incompatible data sets are retrieved, they are mapped into a common representation provided by the meta model. Ideally, the meta model should be based on a theory that contains the theory of all the underlying data models. In practice, most of the meta models for federated database systems are either extended relational or object oriented.

## 2.2. Cyrano Architecture.

Cyrano is a new meta model that can be roughly classified as an object oriented data model. It uses classes as a primary method of structuring the data and resolving heterogeneity. It differs from the object oriented models in that objects of a class must satisfy rules, as opposed to having same attributes. Like object oriented models, Cyrano encapsulates the implementation of data items within objects. Like rule based models, it allows data items to be derived from other data items.

Cyrano has adopted an objected oriented architecture composed of multiple databases arranged in hierarchy. Each of the databases in the hierarchy represents a degree of federation. Figure 1 shows the architecture of Cyrano. Each database in Cyrano is differentiated based on the transactions it processes and emphasizes the behavior instead of the structure. The various databases are described next.

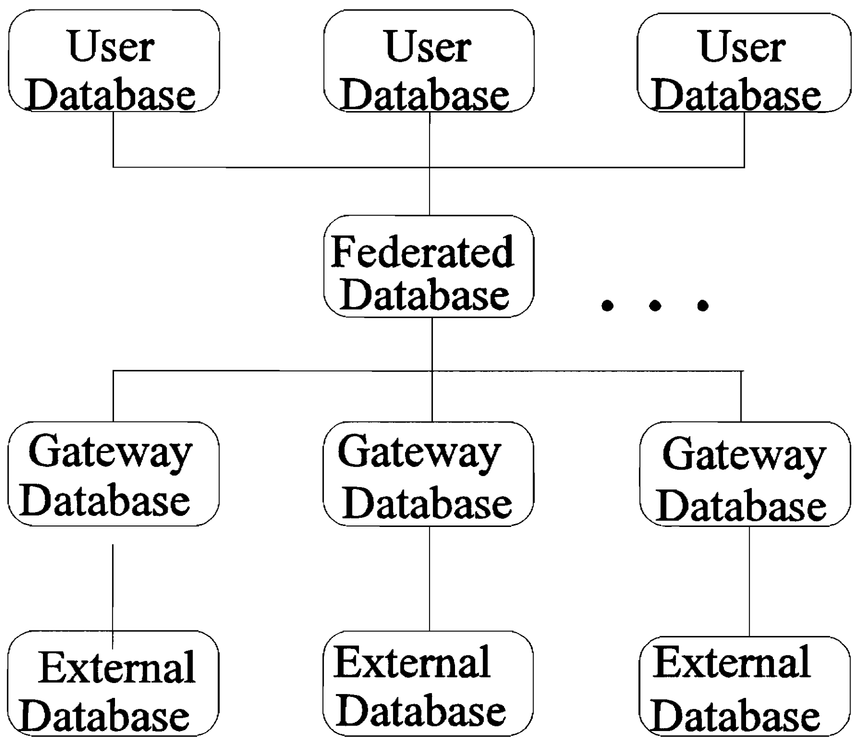


**Figure 1. Cyrano Architecture**

### 2.2.1. External Database.

An External database is the real database with its own DBMS. An External database could be a DBMS based on relational or object oriented or any other model. From the External database's point of view, the FDBMS is seen as an application requesting the data access.

### 2.2.2. Gateway Database.

The objective of the gateway database is to translate the external database representation into an equivalent representation using the meta model constructs. It serves as a gateway between FDBMS and external database. It maps the FDBMS transactions to the external database's own language and forwards them to external database. It also maps the results of transaction back to the FDBMS format. Thus, gateway database provides translations across all possible data models.

### 2.2.3. Federated Database.

A federated database is a federation of a set of multiple gateway databases. It breaks the transactions into sub-transactions and sends them to appropriate gateway database. It also combines the results of sub-transactions to make one result.

### 2.2.4. User Database.

A user database is a user's view of a database. The user database is not specific to the Cyrano architecture and is analogous to views in relational model.

The different databases, arranged in a hierarchy, provide a mechanism for translation across the architectural layers which is a central problem in FDBMS.

## 2.3. Cyrano Classes

In object oriented models, classes are the primary mechanism of structuring data. The class defines the attributes of objects. The objects are created by instantiating classes. Thus, class precedes the object. The schema of object oriented database is formed by the collection of classes and their inter-relation.

Cyrano presupposes a universe containing all possible objects with all possible combinations of attributes. Cyrano classes group the similar objects by a set of defining rules. All the objects that satisfy the rules become members of the class. Thus in Cyrano, object precedes the class, a major departure from traditional notion of class and objects. Cyrano supports three types of classes.

### 2.3.1. Built-in Class.

A Cyrano built-in object is one that is built into the implementation of Cyrano. A built-in object is a member of built-in class. Examples of built in class are Integer and String.

### 2.3.2. Gateway Class.

An external object is one that is stored in the external database. A gateway class is a gateway to the data stored in the external database. A gateway class provides the translation of external database structure into the Cyrano model and is used by Cyrano backend Roxanne.

### 2.3.3 Derived Class.

A Cyrano derived object is one derived from other objects. A derived class groups the derived objects. The derived class can also represent classification of known objects. Derived classes is the mechanism by which Cyrano resolves the database heterogeneity. The rules of the derived class can perform the necessary mapping between different data items.

In traditional object oriented models, the derived class inherits the attributes of the parent class. In Cyrano, the class is a parent to a subclass only if the rules of the subclass imply membership to the parent class. Derived classes are of importance to this project, because they are used in query translation. The Cyrano query is an ad hoc class definition which lists the rules that must be satisfied by external objects to match the query. The original BNF for the derived class is shown in figure 2. This BNF is extended in the sample query translation to accommodate different types of systems. An example of the derived class "OVERSIZED_BOX" is shown below.

```
/* FIRST DERIVATION */
CLASS BOX IS DERIVED  WITH
        /* SUPERCLASS */
        SHAPES  S :
        WITH
                        /* METHODS */
                INT     HEIGHT       =  S.HEIGHT;
                INT     BASE_AREA =  S.BASE_AREA;
        END;
END CLASS.


/* SECOND DERIVED CLASS DEFINITION */
CLASS OVERSIZED_BOX IS DERIVED WITH
        /* BASE OBJECT */
        BOX  B :
                /* GUARD CONDITION */
                B.HEIGHT  > 36
        WITH
```

```
            /* DERIVED METHOD */
            INT BOX_VOLUME = B.HEIGHT * B.BASE_AREA;
     END;
ENDCLASS.
```

A derived class definition consists of one or more sets of derivation rules. Each derivation rule starts with a list of objects and classes to which objects belong. These are the base objects of derived class. In the example for "OVERSIZED_BOX," the base object "B" is the object of the class "BOX."

The second part of the derivation rule is the guard. The guard contains the rule that must be satisfied by base objects before they can be used. In the rule, the name of the base object can appear as free variables. The guard rule evaluates as true or false. If guard evaluates as true, then all combinations of base objects matching the rules belong to the derived class. In the above example, BOXES with "HEIGHT" greater that 36 inches can serve as objects for the "OVERSIZED _BOX" class.

The third part of the derivation is a derived method. A derived method is used to satisfy a message to a derived object. The OVERSIZED_BOX has a method "BOX_VOLUME" defined in terms of the BOX's "BASE_AREA" and "HEIGHT" attributes.

```
<CLASS>
:: =  "CLASS" <NAME>
                    "IS" "DERIVED" "WITH"  <DERIVATION> ";"
                    "END" "CLASS" "."

<DERIVATION>
:: =  <VARIABLE> {"," <VARIABLE>} ":" {<GUARD_VALUE>}
     "WITH" {<DERIVED_METHOD> ";" }
      "END"

<VARIABLE>              :: = <CLASS_NAME> <VAR_NAME>

<CLASS_NAME>            :: = <NAME>

<VAR_NAME>              ::  = <NAME>

<GUARD_VALUE>           ::  =  <COMPLEX_VALUE>

<COMPLEX_VALUE>
:: = <VAR_NAME> "." <NAME><COMP_OPERATOR>  <VAR_NAME>"." <NAME>
|    <VAR_NAME> "." <NAME>{ <COMP_OPERATOR>  <CONSTANT_VALUE>}

<COMP_OPERATOR>   :: =  "=" |  ">" | "<" | "IN" | "<>"| ">="| "<=" |"LIKE"

<DERIVED_METHOD>     :: = <CLASS_NAME> <VAR_NAME> "="  <VALUE>

<VALUE>
:: = <VAR_NAME> "." <NAME>  {<OPERATOR>  <VAR_NAME>"." <NAME>}
|    <CONSTANT_VALUE>

<CONSTANT_VALUE>     :: = <Quated_String> | <Integer> | <Character>

<OPERATOR>              :: =  "+" | "-" | "*" | "/"
```

**Figure 2. Original BNF for Derived Cyrano Class**

# CHAPTER 3.
# FRONTEND FOR CYRANO

Frontend subsystems are defined by [DATE] as families of subsystems that reside on top of DBMS and assist users in solving the problems of information management like storing, accessing, manipulating, and presenting data. They are as important as the core of DBMS because the user interacts with DBMS using these subsystems. Based on the type of functions performed, frontend subsystems are primarily classified into four categories. The four categories are:

- Data access components.
- Data presentation components.
- Application generation components.
- Other components

The data access component deals with the process of locating and retrieving the data. Data presentation refers to the process of displaying data. Application generator allows users to create applications without programming in the traditional sense. Other components include the statistical packages, word processors etc. This project primarily concentrates on the data access component of frontend systems and in particular on the query language. This section first describes a problem definition for a Cyrano frontend and then discusses the approach to solve the problem.

## 3.1 Problem Definition

The scope of Cyrano frontend development considered in this project is limited to the two key problems. The two problems are:

- Defining a query language and a user interface.
- Translating the query into Cyrano classes

These two problems are discussed in following sections.

## 3.2 Query Language and User Interface

The query language is a key component of any database management system. It provides the mechanism for accessing and updating the underlying databases. The literature search for various implementations of federated databases, clearly indicated two popular choices for user interface.

- Query by forms.
- SQL language.

Query by forms though easy for the user to perform, has limitation as far as complex queries are concerned. Also the forms have to be designed first before they can be used to access data.

The SQL language has become the de facto standard for the relational DBMS world. The ANSI standard for SQL 92 is already available and SQL3 work group is in progress. It was felt that SQL, being the more accepted interface language between the two, should be the query language of the Cyrano frontend .

## 3.3 Translation of Query to Cyrano Classes.

The query translation problem was addressed step by step by noting the various features of Cyrano. The key points in this process and the reasoning behind them are described below.

- The Cyrano derived classes are used for accessing the external objects and for resolving the heterogeneity of underlying databases. So the query translation primarily involves the creation of derived Cyrano classes corresponding to the SQL query.

- The derived Cyrano class is created by specifying the base objects and guard rule.

- The base objects are objects that already exist in external databases. A base class is an external class name that is referenced in the SQL query. So the database names following the "FROM" keyword in SQL query can be used as base classes.

- The Cyrano guard rule is used to select the objects belonging to derived class. Thus, the guard rule defines the query conditions. The query condition in SQL is listed in the "WHERE" clause, so the rule should match the "WHERE" clause. The guard definition uses the messages from base classes.

- Cyrano allows the creation of a derivation hierarchy which can be used for matching the sub-queries. Many of the SQL queries can be written in nested sub-query form. The example described in the next chapter uses this mechanism to create a hierarchy of classes matching the query.

- The set of derived query classes are passed to the Cyrano backend Roxanne for actual processing of queries. The results of queries are returned to the frontend which displays them.

Thus, the key component of the frontend development is the translation of the SQL query into one or more derived Cyrano classes.

## 3.4 Problem Approach

Once the high level translation approach was clear, the next step was to create a sample query and perform the manual translation of the query. A set of database schemata was created by Professor Egyhazy for this purpose which was generic enough to accommodate most of the existing data models supported by Cyrano. All the schemata were for the same type of application and stored similar information in different formats.

Next step was to define a set of moderately complex queries. From this set, a query which used all the databases was selected as a sample query. The query in simple English text was transformed into SQL format. Then, one system was considered at a time and the sample query was translated to correspond to that system's model. This involved a trial and error type of a approach. Once all the models were translated, whole process was repeated to resolve the problems encountered. The final translation revealed that the Cyrano class structure needs to be extended to accommodate different systems.

The database schemata for various types of systems and the translation of sample query are discussed in next chapter.

# CHAPTER 4.
# SAMPLE QUERY TRANSLATION

The Database schemata are listed in figures 3 to figure 7. This set consists of two relational schemata, one object oriented schema, one hierarchical schema, and one network schema. Cyrano is able to support all the above mentioned data models. This section describes the heterogeneity in the various schemata, the sample query, the sample query translation and the rationale for the translation.

## 4.1 Heterogeneity in the Schemata.

The database considered in above schemata is STUDENT-COURSE type of database which is used to keep track of courses taken by students and grade received.

Schema 1 shown in figure 3 is relational schema (hereafter referred as Schema 1 Relational) and has three databases, STUDENTS, COURSES, and STUDENT_TAKE_COURSES. The primary keys are indicated at the bottom of each database. Schema 2 shown in figure 4 is also relational (Schema 2 Relational) and is different from Schema 1 Relational in database names and attribute names. For example, social security number is referred to as "SS#" in Schema 1 Relational and "STUD_ID" in Schema 2 Relational. The attribute "NAME" is duplicate in STUDENTS and CLASSES databases in Schema 2 Relational. In STUDENTS database it refers to student's name and in CLASSES database it refers to course title.

These examples show that even if two databases are based on the same model they could be heterogeneous because of difference in attribute and database names. Another kind of heterogeneity present in these two databases is due to the difference in attribute types. For example, GRADE attribute in Schema 1 Relational is NUMERIC while GRADE in Schema

2 Relational is CHAR. The attribute type heterogeneity is resolved at the federated database level. This falls outside the scope of this project.

The heterogeneity arising due to the difference in names is resolved in the frontend translation as attributes and database names are specified in the query. It is more appropriate to resolve it at the frontend level because the frontend can establish a dialog with the user in case of doubt. For example, if user enters "SELECT NAME" and does not specify the FROM clause, then query is ambiguous because the attribute" NAME" exists in two databases (STUDENT and CLASSES) in Schema 2 Relational. In such a case the frontend prompts the user saying that attribute name is ambiguous and the user needs to give more information. Another advantage is that the frontend can validate the query before passing to the backend.

Figure 5 shows the partial schema for STUDENT-COURSES databases in object oriented format (Schema 3 Object Oriented). The three class definitions represent the object oriented representation of schemata. The class COURSE_STUDENT is derived from COURSE class and STUDENT class. Figure 6 shows the hierarchical structure (Schema 4 Hierarchical) in IMS format and figure 7 shows the network structure(Schema 5 Network) in IDMS format. The individual schemata are not discussed here as their syntax are not relevant to this project.

**TABLE**:STUDENTS

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| SS# | NUMERIC | 9 | NOT NULL |
| NAME | CHAR | 25 | NOT NULL |
| BIRTHDATE | NUMERIC | 6 | NULL |
| SEX | CHAR | 1 | NULL |
| PHONE | NUMERIC | 10 | NULL |

**PRIMARY KEY** (SS#)

**TABLE**:COURSES

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| INDEX | NUMERIC | 7 | NOT NULL |
| NUMBER | ALPHANUM | 6 | NOT NULL |
| TITLE | CHAR | 25 | NULL |

**PRIMARY KEY** (INDEX)

**TABLE**:STUDENT_TAKE_COURSES

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| SS# | NUMERIC | 9 | NOT NULL |
| INDEX | NUMERIC | 7 | NOT NULL |
| GRADE | NUMERIC | 3 | NOT NULL |

**PRIMARY KEY** (SS#, INDEX)

**Figure 3. Schema 1 Relational System**

**TABLE**:STUDENTS

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| STUD_ID | NUMERIC | 11 | NOT NULL |
| NAME | CHAR | 30 | NOT NULL |
| BIRTHDATE | NUMERIC | 4 | NULL |
| SEX | CHAR | 6 | NULL |
| ADDRESS | ALPHANUM | 35 | NOT NULL |

**PRIMARY KEY** (STUD_ID)

**TABLE**:CLASSES

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| INDEX | NUMERIC | 7 | NOT NULL |
| COURSES | ALPHANUM | 6 | NOT NULL |
| NAME | ALPHANUM | 25 | NULL |

**PRIMARY KEY** (INDEX)

**TABLE**:CLASS_ENROLLMENTS

**ATTRIBUTES**

| **NAME** | **TYPE** | **LENGTH** | **EXIST COND.** |
|---|---|---|---|
| STUD_ID | NUMERIC | 11 | NOT NULL |
| INDEX | NUMERIC | 7 | NOT NULL |
| GRADE | CHAR | 1 | NULL |

**PRIMARY KEY** (STUD_ID, INDEX)

**Figure 4. Schema 2 Relational System**

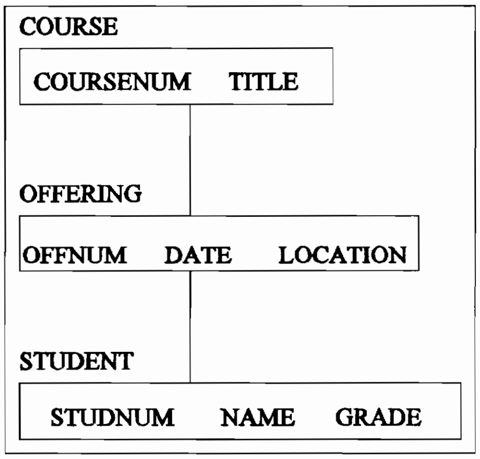
```
define Type COURSE
        attributes = {
                        INDEX : Identifier
                        NAME : optional String
                     }
        methods =  {
                        display: COURSE info
                     }

define Type STUDENT
        attributes = {
                        STUD_ID      : Identifier
                        NAME         : optional String
                        BIRTHDATE : optional String
                        SEX           : optional Character
                        ADDRESS    : required String
                     }
        methods =  {
                        display: STUDENT info
                     }

define Type COURSE_STUDENTS
        supertype =     {STUDENTS}
        supertype =     {COURSE}
        attributes = {
                        GRADE         :  required Character
                     }

        methods = {
                        list: STUDENT'S in COURSE
                    }
```

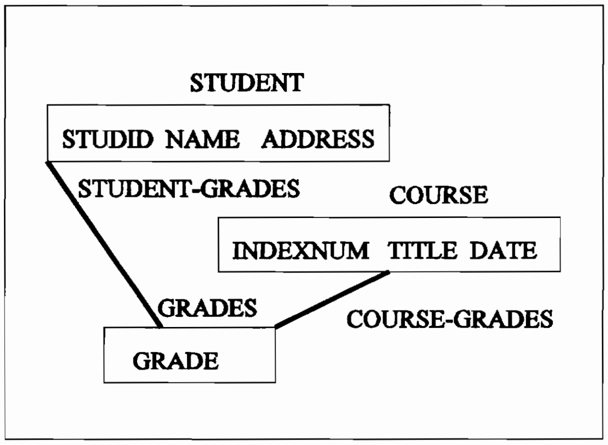
**Figure 5. Schema 3 Object Oriented System**



HIERARCHICAL STRUCTURE SCHEMA

| | | |
|---|---|---|
| 1. | DBD | NAME=EDUCDDB |
| 2. | SEG | NAME=COURSE, BYTES=39 |
| 3 | FIELD | NAME=(COURSENUM,SEQ), BYTES=6,START=1 |
| 4. | FIELD | NAME=TITLE,BYTES 33,START 7 |
| 5. | SEG | NAME=OFFERING,PARENT=COURSE, BYTES=25 |
| 6. | FIELD | NAME=(OFFNUM,SEQ), BYTES=7, START=1 |
| 7. | FIELD | NAME=DATE,BYTES=6,START=8 |
| 8. | FIELD | NAME=LOCATION,BYTES=12,START=14 |
| 9. | SEG | NAME=STUDENT,PARENT=OFFERING,BYTES=37 |
| 10. | FIELD | NAME=(STUDNUM,SEQ),BYTES=6,START=1 |
| 11. | FIELD | NAME=NAME, BYTES=30,START=7 |
| 12. | FIELD | NAME=GRADE,BYTES=1,START=37 |

**Figure 6. Schema 4 Hierarchical System**



```
1       SCHEMA  NAME IS GRADEEREPORT
2.      RECORD NAME IS STUDENT
3.      LOCATION MODE IS CALC USING STUDID DUPLICATES NOT ALLOWED
4.              02      STUDID      PIC     X(9)
5.              02      NAME        PIC     X(25)
6.              02      ADDRESS     PIC     X(35)
7.      RECORD NAME IS COURSE
8.      LOCATION MODE IS CALC USING INDEXNUM
                                    DUPLICATES NOT ALLOWED
9.              02      INDEXNUM    PIC     X(7)
10.             02      TITLE       PIC     X(25)
11.             02      DATE        PIC     X(6)
12.     RECORD NAME IS GRADES
13.     LOCATION MODE IS VIA STUDENT-GRADES SET
14.             02      GRADE       PIC     X(1)
15.     SET NAME IS STUDENT-GRADES
16.     ORDER IS LAST
17.     OWNER IS STUDENT
18.     MEMBER IS GRADES MANDATORY AUTOMATIC
19.     SET NAME IS COURSE-GRADES
20.     ORDER IS SORTED
21.     OWNER IS COURSE
22.     MEMBER IS GRADES OPTIONAL  MANUAL
```

**Figure 7. Schema 5 Network System**

## 4.2 Sample SQL Query

The sample query is for Schema 1 Relational and it accesses all the three databases. The query in English language and SQL form are shown below.

- **QUERY**

*List social security number and name of all the students that are in course that has "DATABASE" as one of the words in the title.*

- **SQL QUERY**

```
SELECT SS#, NAME
FROM STUDENTS
WHERE SS# IN
        (SELECT SS#
         FROM STUDENTS_TAKE_COURSES
         WHERE INDEX IN
                 (SELECT INDEX
                  FROM COURSES
                  WHERE TITLE LIKE '%DATABASE%')))
```

The query was arranged in sub-query format as it leads to easier translation as discussed in section 3.3.

## 4.3 Translation of Sample Query to Cyrano Classes

```
/* Corresponds to the innermost sub-query. */
CLASS INNER_1 IS DERIVED WITH

                /* Base objects. Resolve database name heterogeneity */
                COURSES A          OR      /*Schema 1 Relational */
```

```
          CLASSES  B         OR    /*Schema 2 Relational */
          COURSE   C         OR    /*Schema 3 Object oriented */
          COURSE   D         OR    /*Schema 4 Hierarchical */
          COURSE   E               /*Schema 5 Network */

     :    /* Guard Rule. Resolve attribute name heterogeneity. */
          A.TITLE            OR
          B.NAME             OR
          C.NAME             OR
          D.TITLE            OR
          E.TITLE

     /* The query condition in WHERE clause. */
     LIKE "%DATABASE%"
     WITH
          /* Derived method. */
          INT INDEX =        A.INDEX          OR
                             B.INDEX          OR
                             C.INDEX          OR
                             D.COURSENUM      OR
                             E.INDEXNUM       ;
     END;
ENDCLASS.

/* Corresponds to second innermost sub-query. */
CLASS INNER_2 IS DERIVED WITH

          /* First derivation */
          STUDENT_TAKE_COURSES       A      OR
          CLASS_ENROLLMENTS          B      OR
          COURSE-STUDENTS            C      OR
          OFFERING                   D      OR
          GRADES                     E      ,
          INNER_1                    F   /* Derivation from INNER_1 */
   :
          A.INDEX                    OR
          B.INDEX                    OR
          C.COURSE.INDEX             OR
          D.NIL                      OR  /* Hierarchical system needs no
                                            further mapping. */
          E.MEMBER(COURSE)               /*Network Schema */
```

```
        IN
                F.INDEX
        WITH
                /* Derived Method. */
                INT SS# =
                                A.SS#                   OR
                                B.STUD_ID               OR
                                C.STUDENT.STUD_ID       OR
                                D.NIL                   OR
                                E.OWNER(GRADE)          ;
        END;
ENDCLASS.

/* The outermost sub-query translation. */
CLASS OUTER IS DERIVED WITH

                /* First derivation */
                STUDENTS  A                     OR
                STUDENTS  B                     OR
                STUDENT   C                     OR
                STUDENT   D                     OR
                STUDENT   E                     ,
                INNER_2   F                     /* Derivation from INNER_2 */
       :
                A.SS#                           OR
                B.STUD_ID                       OR
                C.STUD_ID                       OR
                D.NIL                           OR
                E.MEMBER(STUDENT)
       IN
                F.SS#
        WITH
                /* Derived methods  corresponding to SELECT clause */
                INT SS# =
                                A.SS#           OR
                                B.STUD_ID       OR
                                C.STUD_ID       OR
                                D.STUDNUM       OR
                                E.STUDID        ;
                STRING NAME =
                        A.NAME          OR
                        B.NAME          OR
                        C.NAME          OR
                        D.NAME          OR
                        E.NAME          ;
          END;
ENDCLASS.
```

## 4.4 Rationale for Sample Query Translation

In the sample query translation, each derived class corresponds to the sub-query specified in the SQL format. The translation starts from the innermost query and proceeds outward till the outermost sub-query is translated. The class derived in the inner sub-query is used in the derivation of the outer query. The set of classes generated during this process is the Cyrano representation of the query.

The first class INNER_1 translates innermost sub-query. The base object is the COURSES database with the rule matching condition specified in the "WHERE" part. The COURSES database has different names in different schemata so an "OR" clause is used to resolve the difference in names of base object. Here the key assumption is that the query translator has access to the federated schema mapping. Such a schema mapping would indicate that names such as COURSES, CLASSES, COURSE are synonyms as far as FDBMS is concerned as they correspond to similar databases in underlying DBMSs. A similar mapping is assumed for attribute names which shows that TITLE in Schema 1 Relational is same as NAME in Schema 3 Object Oriented. Such interface is assumed to exist.

The class INNER_2 is an example of multiple derivation and is derived from class INNER_1 and STUDENT_TAKE_COURSES database. This class corresponds to the second innermost sub-query in SQL format. The guard rule for this class joins a base class and the class INNER_1 derived earlier. The method part of this class defines the variable SS# which is used to store the social security number for the objects that belong to this class. The type "INT" for the SS# variable is assumed to be known. The rule in INNER_2 has "NIL" for Schema 4 Hierarchical because once COURSENUM is identified, all the underlying OFFERING records are accessible in the hierarchical database. Similarly, the rule for Schema 5 Network has term MEMBER(COURSE). This indicates a link corresponding to MEMBER for this record. The member can be located by accessing the COURSE_GRADE set. Cyrano backend will have to be intelligent enough to infer such meanings. The method part for Schema 5 Network shows a similar OWNER term which indicates the owner link corresponding to the GRADE record. This part of the translation is not easy enough and needs to be redesigned based on backend implementation. This shows that the translation for non relational or non object oriented system is not easy. The primary reason for this is that, Cyrano is object oriented and the query interface is relational.

The class OUTER is derived from class INNER_2 and the STUDENTS base class. The class OUTER corresponds to the outermost SQL sub-query. The guard rule performs the join of two classes. The derived method defines two variables corresponding to the "SELECT" clause. These two variables specify the expected output (SS# and NAME ) of the query.

8.

# CHAPTER 5.
# FRONTEND PROTOTYPE

The prototype primarily concentrated on the query translation part of frontend, rather than user interface. To provide the proof of concept, the prototype development limited itself to the query described in the previous section. The prototype design is divided into three parts.

- Parser.
- Lexical Analyzer.
- Cyrano Translator.

Each of these parts is discussed next.

## 5.1 Parser

The grammar for the SQL supported by the prototype is as shown below:

```
1.   expression    -----> |-
                     |    sel_term from_term where_term

2.   sel_term      -----> SELECT s_term sel_term'

3.   sel_term'     -----> , s_term sel_term'
                     |    ϵ

4.   from_term     -----> FROM s_term

5.   where_term    -----> WHERE s_term comp_term where_term'

6.   where_term'   -----> ( expression)
                     |    s_term

7.   s_term        -----> VALID_TERM

8.   comp_term     -----> VALID_COMP_TERM
```

The production 1 shows the basic SQL expression of SELECT-FROM-WHERE type. This basic expression is the core of the sub-query. Production 2 and 3 show the format of SELECT clause. Select clause has keyword SELECT followed by attribute s_term. Multiple attributes can be specified in select clause each separated by comma. Production 4 describes the FROM clause. For simplicity the FROM clause accepts only one database name but it can be easily extended. Production 5 and 6 show the WHERE clause. The WHERE clause has two formats. One format allows the comparison of two terms. The second format allows to write nested sub-queries. A sub-query is similar to the query and is an expression of type SELECT-FROM-WHERE. The grammar described here is sufficient to evaluate the sample query described earlier.

## 5.2 Lexical Analyzer

The lexical analyzer for the prototype recognizes the tokens shown in table 1.

**TABLE 1. Lexical Analyzer Tokens**

| Tokens | Lexemes | Description |
|---|---|---|
| S_TERM | [a-zA-Z0-9], " _", " ' ", " " ", "#", "%" | A word with valid letters |
| COMP_TERM | < , > , = , IN, LIKE | A comparison term |
| SEP_TERM | " , " | A Comma separator |
| LFT_PAR_TERM | ( | Left parenthesis |
| RGT_PAR_TERM | ) | Right parenthesis |
| SELECT_TERM | SELECT | Key word |
| FROM_TERM | FROM | Key word |
| WHERE_TERM | WHERE | Key word |
| EOI | NULL | End of input |

The "SPACE " is used as white space character. The input string is scanned till a valid token is detected or a white space is encountered. The term collected is classified into appropriate token.

## 5.3 Cyrano Translator

The Cyrano translator translates the sample query into Cyrano classes which are saved in text format in a output file. The translator is divided into various functions that use the terms gathered by the parser. It then generates the various parts of derived Cyrano class. The translator does not have the data dictionary interface, so it does not validate the database names. The translator has a built-in table which stores the data type of attributes specified in the sample query. For example, the data type of attribute TITLE is "STRING" and is stored in the built-in table.

# CHAPTER 6.
# FRONTEND IMPLEMENTATION

This section discusses some of the key implementation details at a higher level. The code is listed in the appendix. The Cyrano backend Roxanne is being developed in Borland C++. In order to facilitate easy merging, it was decided to use Borland C++ for the frontend. The Borland C++ version 3.0 compiler running under DOS was used to build the prototype. The code for the lexical analyzer and the parser for the prototype were reused from the [HOLUB]. It was modified to suit the frontend design. The code was documented using the guidelines from [FRAKES].

## 6.1 Parser Implementation

The parser is a primitive recursive descent expression parser and is listed in the CYR_GRA.C file. Each subroutine in the parser code corresponds to the left hand side of the grammar specified in last section. For example, the production

*from_term* -----> **FROM** *s_term*

is implemented as

```
void from_term(char * fr_term)
{
 if (match(FROM_TERM) {
        advance();
        sterm(fr_term);
    }
}
```

Some of the functions in the parser, such as *where_term* are recursive and allow the nested sub-queries to be processed. In order to support the recursion, the variables used in these functions are automatic variables so that they are allocated on the stack. The parser validates the input and if the input does not match the grammar then it rejects the query.

In addition to validating the input, the parser acts as a term collector. The various terms in SQL clauses such as database names are collected by the parser. The *expression* is the highest level parser function and it has arrays to collect the terms. The *expression* being recursive, the local arrays are allocated each time a nested expression is encountered. The pointers to the term collecting arrays are passed as function arguments to lower level parser functions. These arrays are used by the Cyrano translator to generate the derived classes.

## 6.2 Lexical Analyzer Implementation

The lexical analyzer is listed in the CYR_LEX.C file. The token identification is performed at two levels. First a single character is analyzed to see if it is a valid token. The example of single character token is " = " or ">" character. If the character is not a token then input is processed until a white space is encountered and the complete term is collected. The term collected is then compared with valid keywords and if matched, the corresponding token is returned. The lexical analyzer is implemented in a single function.

## 6.3 Cyrano Translator

The Cyrano translator uses the terms collected by parser to create the Cyrano classes. The Cyrano translator is a collection of functions which generate the derived Cyrano classes. The translator is invoked only if the parser does not detect an error. The prototype translator generates Cyrano classes in ASCII text format and writes the complete set of classes into a file "OUT_FILE". The classes generated can be viewed by displaying output file.

The translator generates the derived class names automatically. The first class generated is named "CL_0", and second class generated is named "CL_1" etc. The internal counter "class_no" is used to generate the names and is incremented after each class generation. The translator also uses an internal character array to generate the base object names. The base object names are single character and increase in alphabetical order. For example, the first base object corresponding to the COURSES database is named as 'A'.

The built-in data dictionary in the prototype stores the data types for the attributes. The prototype does not have any mapping of the databases under different systems. As a result the prototype generates the output for only one type of database. The output generated by prototype for the sample query is shown below.

```
CLASS CL_0 IS DERIVED WITH
        COURSES A : A.TITLE  LIKE  "%DATABASE%"
WITH
        INT INDEX = A.INDEX
        END
ENDCLASS

CLASS CL_1 IS DERIVED WITH
        STUDENTS_TAKE_COURSES B, CL_0 C :  B.INDEX  =  C.INDEX
WITH
        INT SS# = B.SS#
        END
ENDCLASS
```

```
CLASS CL_2 IS DERIVED WITH

        STUDENTS D, CL_1 E : D.SS#  =  E.SS#
WITH
        INT SS# = D.SS#
        END
ENDCLASS
```

# CHAPTER 7.
# CONCLUSIONS

A good user interface with a strong query language is a necessary component of any database management system. The SQL is standard for commercial relational databases and has a large following of users. So the choice of SQL as the query language for Cyrano appears to be quite appropriate.

Cyrano supports relational, network, hierarchical, object oriented and rule based families of data models. For relational or object oriented systems the query translation is simple. However, the query translation for hierarchical or network system is complex, as shown in the sample query translation in this paper. More research is required to refine the translation. Also more complex queries involving multiple databases should be translated.

The sample query translation gives a proof of concept to map relational queries into object oriented and rule based Cyrano representations. The prototype can be looked as a demonstration of the feasibility of the concept. The heterogeneity in database and attribute names can be efficiently resolved at the frontend level by use of an "OR" clause. This project demonstrates that, a user interface based on SQL is a feasible option for specifying a derived class ( i.e. a query) in the Cyarno.

The prototype implementation can be extended in many ways. The lexical analyzer can be replaced with the lex and the parser with the yacc. The grammar can be extended to provide full support to the SQL data manipulation language. The user interface can be upgraded to WINDOWS platform. Finally, the output of the Cyrano frontend can be linked to the backend Roxanne, so that a user can execute real queries.

# BIBLIOGRAPHY

Ahmed R., et al, The Pegasus Heterogenous Multidatabase System, Computer, Vol 24, NO 12, December, 1991, p. 19-27

Breitbart Y., Olson P. L., and Thompson G. R., Database Integration in a Distributed Heterogenous Database System, Proceedings of the International Conference on Database Engineering, IEEE, Washington DC, 1986, p.301-310

Chung C., DATAPLEX: An access to Heterogenous Distributed Databases, Communications of the ACM, Vol 33, No 1, January, 1990, p. 70-80

Date C. J.,An Introduction to Database Systems Vol. I, Addison-Wesley Publishing Company, New York, 1991

Dzikiewicz Joseph, Csaba Egyhazy, Cyrano : A Meta Model for Federated Database Systems, Doctoral dissertation in progress, Virginia Polytechnic and State University, VA 1994

Frakes William, Fox Christopher, Nejmeh Brian, Software Engineering in the Unix/C environment, Prentice Hall, Englewood Cliffs, New Jersey, 1991

Holub Allen, Compiler Design in C, Prentice Hall Englewood Cliffs, New Jersey, 1990

Ram S., Heterogenous Distributed Database Systems, Computer, Vol 24, No 12, December, 1991, p. 7-11

Shipman D., The Functional Data Model and the Data Language DAPLEX, ACM Transactions on Database Systems, Vol 6, NO 1, March, 1981 p. 140-173

Thomas G, Thompson G.R., Chung C.W, Barkmeyer E., Carter F., Templeton M., Fox S.,Hartman B., Heterogenous Distributed database Systems for Production Use, ACM Computing Surveys, Vol 22, No 3, September, 1990, p. 237-266

# APPENDIX A - SOURCE CODE

```
/*****************     Cyrano.h     *********************************

        Purpose :     To define the constants used in Cyrano Frontend.

        History :

        Date          Name                      Comment

        10/15/94      NSamant                   Created the header file.

*******************************************************************/

/******************     Public Definition     *************************/
#define FALSE         0
#define TRUE          1

/******************     Enumeration constants for tokens     ************/
enum token_type {
                      EOI,                      /* End of input            */
                      S_TERM,                   /* A valid term            */
                      SELECT_TERM,              /* SELECT keyword          */
                      FROM_TERM,                /* FROM keyword            */
                      WHERE_TERM,               /* WHERE keyword           */
                      COMP_TERM,                /* Comparison term >,= etc. */
                      LFT_PAR_TERM,             /* Left brace              */
                      RGT_PAR_TERM,             /* Right brace             */
                      SEP_TERM                  /* A separator term ","    */

        };

/******************     Data Dictionary structure     *****************/

typedef struct {
        char * attrib_name;             /* Attribute name          */
        char * attrib_type;             /* Attribute Structure     */
        } DATA_DICT_REC;

/*****************     End of Cyrano.h     *************************/
```

```
/*******************     cyr_lex.c     ******************************
  Purpose:   Parse the input and match the terms to token.

  Notes :    This module contains a lexical analyzer. The lexical
             analyzer scans the input string for valid tokens.
             The token is returned to parser. The term is stored
             in global variable cur_lexme.

  History:

  Date       Name                        Comment
  10/15/94   NSamant                     Created the file.
*******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "cyrano.h"            /* Frontend specific definitions */

/*******************     Private variables         ********************/

static char *yytext = "";          /* Keeps track of current lexeme */
static int  yylength =0;           /* No of characters in lexeme      */

/*******************     Private functions         ********************/

static int is_valid_char(int c);

/*******************     Public variables          ********************/

extern char input_buffer[256];      /* Input buffer            */
char cur_lexeme[32];                /* The current valid term  */

/*******************     Public functions          ********************/

int cyr_lex(void);

/*******************     Private functions         ********************/

/******************************************************************
             is_valid_char(int c)
```

```
Returns :       int

Purpose :       Check if c is valid character.

Plan    :       C is valid if ('_', '\"', '#', '%', '\'')
                or a alphanumeric character.

Notes  :        None
*****************************************************************/

static int is_valid_char(int c)
{

if (c == '_' || c == '\"' || c == '#' || c == '%' || c == '\'')
        return(TRUE);
else
        return(isalnum(c));
}

/*****************        Public functions        ******************/
/*****************************************************************
                cyr_lex(void)

Returns :       int

Purpose :       To classify the terms into different tokens.
                Also copies the term into global variable cur_lexeme.

Plan    :       Skip all the blanks.
                Check the next character.
                If it is token character then return token.
                If not, get the full term.
                Match the term and return the token.

Notes  :        None
*****************************************************************/
int cyr_lex(void)
{

char * current;         /* Pointer to the current character */
static first_time = 1;  /* Flag to indicate first time function is called */
```

```
/* Initialize current to current location in input buffer. */
current = yytext + yylength;
memset(cur_lexeme,0,32);

/*If first time set current to input buffer and set flag false. */
if(first_time)  {
        current = input_buffer;
        first_time = 0;
        }

/* If end of input */
if(!*current)
        return(EOI);

while(isspace(*current)) /* Skip spaces. */
        ++current;

for(;*current;++current) {
        yytext = current;
        yylength = 1;

        /* Copy the character into cur_lexeme. */
        cur_lexeme[0] = *current;
        cur_lexeme[1] = '\0';
        switch(*current) {      /* First check character. */

                case '\0': return EOI;
                case '=' : return COMP_TERM;
                case '>' : return COMP_TERM;
                case '<' : return COMP_TERM;
                case '(' : return LFT_PAR_TERM;
                case ')' : return RGT_PAR_TERM;
                case ',' : return SEP_TERM;
                case '\n': return EOI;
                case '\t':
                case ' ': break;

                default: /* Not a character token. */

                        /* Check if valid characters in term */
                        if(!is_valid_char(*current))
                        printf("Ignoring invalid characters\n");
                else {

                        /* Collect the term till valid character. */
                        while(is_valid_char(*current))
                                ++current;
                        yylength = current - yytext;

                        /* Copy the term into cur_lexeme. */
                        strncpy(cur_lexeme,yytext,yylength);

                        /* Check if term is a keyword.
                         * If yes then return the token.
                         */
                        if(!strcmp(cur_lexeme,"SELECT"))
                                return(SELECT_TERM);
                        else if (!strcmp(cur_lexeme,"FROM"))
                                return(FROM_TERM);
                        else if(!strcmp(cur_lexeme,"WHERE"))
                                return(WHERE_TERM);
                        else if(!strcmp(cur_lexeme,"IN"))
                                return(COMP_TERM);
                        else if(!strcmp(cur_lexeme,"LIKE"))
                                return(COMP_TERM);
                        else
                                /* Otherwise a regular term. */
                                return(S_TERM);
                }
        break;
        }
    }
/* If we reach here then its end of input. */
return(EOI);
}
```

```
/*******************        cyr_cls.c      ******************************

   Purpose:    This module contains routines for Cyrano translator.

   Notes  :    The set of routines create derived Cyrano classes and
               write them to a text file named OUT_FILE.

   History:

   Date        Name                               Comment
   11/01/94    NSamant                            Created the file.
*********************************************************************
/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cyrano.h"         /* Frontend specific definitions */


/*******************        Private variables          *********************/

/* The names for base class are single characters. */
static char class_char[26] = { 'A','B','C','D','E','F','G','H',
                    'I','J','K','L','M','N','O','P',
                    'Q','R','S','T','U','V','W','X',
                    'Y','Z'
                 };

/* Stores the attribute name and data type. Simulates data dictionary.
*  The last element MUST be NULL as search routine uses it to terminate
*  the loop.
*/
static DATA_DICT_REC data_dictionary[100] =  {
                          {"SS#" ,  "INT"}, {"NAME",  "STRING"},
                          {"BIRTHDATE",  "INT"}, {"SEX",  "CHAR"},
                          {"PHONE",  "INT"}, {"INDEX",  "INT"},
                          {"NUMBER",  "STRING"},{"TITLE", "STRING"},
                          {NULL,      NULL}
                          };
```

```
static class_char_p = -1; /* Used to name base class */
static int class_no = 0;   /* Used to create the derived class name */
static FILE * rox_file;   /* OUT_FILE pointer */

/*******************        Private functions         ********************/
static char * find_attr_type(char * attrib_name);

/*******************        Public variables          ********************/

/*******************        Public functions          ********************/
void init_op_file(void );
void close_op_file(void);
void wr_class_name(void );
void wr_base_names(char * from_term);
void wr_guard_rule(char * wh1_term, char *wh_comp_term, char * wh2_term);
void wr_methods(char * sel_term);
void wr_end_class(void);

/*******************        Private functions         *******************/
/******************************************************************
                find_attr_type(char * attrib_name )

Returns :       char *

Purpose :        find the data type of attribute.

Plan    :       Loop through data dictionary array till attribute
                name is matched or NULL is found.

Notes  :        Data dictionary must have last element as NULL
                otherwise loop will never end in case there is no match.
*****************************************************************/
char * find_attr_type(char * attrib_name)
{
int index = 0;  /*Index into data dictionary array */

while ((strcmp(data_dictionary[index].attrib_name,attrib_name)) &&
        data_dictionary[index].attrib_name != NULL)
    index++;

/*Return NULL or matching data type */
```

```
return(data_dictionary[index].attrib_type);
}

/*******************        Public functions        ********************/
/*********************************************************************
                wr_class_name(void)

Returns :       void

Purpose :       To create a new base class name.

Plan    :       Use a standard prefix "CL_".
                Append the class_no in string format to "CL_".
                Increment the class_no.

Notes   :       None
*********************************************************************/
void wr_class_name(void)
{
char temp_buf[40];
fprintf(rox_file,"\n");
sprintf(temp_buf,"CLASS CL_%d IS DERIVED WITH\n",class_no);
fprintf(rox_file,temp_buf);
class_no++;
}
/*********************************************************************
                wr_base_names(char * from_term)

Returns :       void

Purpose :       To create a base object and base class

Plan    :       Use the class_char array to create base object
                name and use database name as base class.
                Use the term following FROM keyword.

Notes   :       None
*********************************************************************/
void wr_base_names(char * from_term)
{
/* Print the first base object and base class name.
```

```
 * The base class is same as database name.
*/
class_char_p++;
fprintf(rox_file,"\t%s %c", from_term,class_char[class_char_p]);

/* If more than one class then print other base class.*/
if(class_no > 1) {
        class_char_p++;

        /* Print the second base object and base class name.
         * The base class is a class already derived.
        */
        fprintf(rox_file,", CL_%d %c",(class_no-2),
                        class_char[class_char_p]);
        }
/* Print the guard separator */
fprintf(rox_file," :");
}

/****************************************************************
            wr_guard_rule(char * wh1_term       First WHERE term
                    char * wh_comp_term        Comparison term
                    char * wh2_term            Second WHERE term
                    )

Returns :   void

Purpose :   To create the guard rule condition.

Plan    :   The guard rule condition is same as WHERE clause
            for innermost class. The guard rule for other classes
            use method of previously derived class.

Notes  :    None
****************************************************************/
void wr_guard_rule(char * wh1_term, char *wh_comp_term, char * wh2_term)
{
/* Rule for innermost class is same as WHERE clause. */
if(class_no < 2)
        fprintf(rox_file,"  %c.%s %s %s\n",class_char[class_char_p], wh1_term,
                    wh_comp_term, wh2_term);
```

```
else
        /* Rule for other classes include  method of previously
         * derived class.
         */
        fprintf(rox_file,"  %c.%s = %c.%s\n",class_char[class_char_p -1],
                        wh1_term,class_char[class_char_p],wh1_term);

}
/*****************************************************************
                wr_methods(void)

Returns :       void

Purpose :       To create a method for derived class .

Plan    :       Use the SELECT term in method.
                Find the matching data type.

Notes  :        None
*****************************************************************/

void wr_methods(char * sel_term)
{
char * var_type;

/* Find the matching data type for attribute. */
if ((var_type = find_attr_type(sel_term) ) == NULL) {
        printf("No matching attribute type for %s in data dictionary\n",
                sel_term);
        var_type = "UNDEFINED";
        }

/* For first derived class use same base object. For others
 * use previous object name.
 */
fprintf(rox_file,"WITH\n");
if(class_no < 2)
        fprintf(rox_file,"\t%s %s = %c.%s;\n",var_type,sel_term,
                 class_char[class_char_p],sel_term);
else
        fprintf(rox_file,"\t%s %s = %c.%s;\n",var_type,sel_term,
                    class_char[class_char_p-1],sel_term);

}
/*****************************************************************
                    wr_end_class(void)

Returns :           void

Purpose :           Create end class marks.

Plan    :           Write END and END CLASS statements.

Notes  :            None
*****************************************************************/

void wr_end_class(void)
{

fprintf(rox_file,"\tEND;\n");
fprintf(rox_file,"ENDCLASS.\n");

}

/*****************************************************************
                    init_op_file(void)

Returns :           void

Purpose :           Open a file "OUT_FILE".

Plan    :           Use standard fopen function.

Notes  :            None
*****************************************************************/
void init_op_file(void )
{

if ( (rox_file =fopen("out_file","w")) == NULL) {
          printf("Enable to create rox_out file.\n");
          exit(1);
          }
```

```
}
/*****************************************************************
                close_op_file(void)

Returns :       void

Purpose :       To close the OUT_FILE file.

Plan    :       Print the output file name.
                Close the file.
Notes   :       None
*****************************************************************/
void close_op_file(void)
{
printf ("\n*****************************************************\n");
printf("The Derived classes have been written to OUT_FILE file\n");
printf ("*****************************************************\n");
fclose(rox_file);
}
```

```
/*******************      cyr_gra.c      *****************************

   Purpose:    Check the grammar.
               Collect the input.
               Also has main function.

   Notes  :    The parser checks the grammar. Each term on
               left hand side of grammar corresponds to function.
               Some functions are recursive to process the nested
               expressions.

   History:

   Date        Name                          Comment
   10/21/94    NSamant                       Created the file.
*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "cyrano.h"             /* Frontend specific definitions */

/*******************      Private variables      *******************/
static int debug_flag = 0;          /* Disable the debugging              */
static int no_errors = FALSE;       /* Flag set if errors in parsing.     */
static int sel_term_p = 0;          /* An index into SELECT terms array.  */
static int lookahead = -1;          /* Stores next token                  */
/* Debugging information */
static char * tok_strings [] = {
                         "EOI","S_TERM", "SELECT_TERM",
                         "FROM_TERM","WHERE_TERM","COMP_TERM",
                           "(_TERM",")_TERM","SEPERATOR_TERM"};
/*******************      Private functions      *******************/
static void advance(void);
static int match( int token);
static void statements(void);
static void expression(void);
static void s_term(char * pres_term);
static void select_term(char sel_term[][32]);
static void from_term(char *fr_term);
static void where_term(char *wher1_term,char *wher_comp_term,
                        char *wher2_term);

/*******************     Public variables          ********************/
char input_buffer[256];          /* Input buffer to hold input query.   */
extern char cur_lexeme[32];      /* The current lexeme.                 */

/*******************     Public functions          ********************/
extern int cyr_lex(void);
extern void init_op_file(void );
extern void close_op_file(void);
extern void wr_class_name(void );
extern void wr_base_names(char * from_term);
extern void wr_guard_rule(char * wh1_term, char *wh_comp_term, char * wh2_term);
extern void wr_methods(char * sel_term);
extern void wr_end_class(void);

/*******************     Private functions         ********************/
/*********************************************************************
                advance(void)

Returns :       void

Purpose :       To advance to the next term.

Plan    :       Call cyr_lex to parse next term from input.
                The token returned is stored in lookahead.

Notes  :        None
*********************************************************************/
static void advance(void)
{
lookahead = cyr_lex();
if(lookahead != EOI) {
        if(debug_flag > 1)
                printf ("Token for %s is %s\n",cur_lexeme,tok_strings[lookahead]);
        }
}
/*********************************************************************
                match(int token)
```

```
Returns :       int

Purpose :       Check if next token matches the one expected.

Plan    :       Compare lookahead with token and return the result.
                If this is first call to match then call cyr_lex
                first, because lookahead is not initialized yet.

Notes  :        None
******************************************************************/

static int match( int token)
{
/* On first match call lookahead is -1 so call cyr_lex() */
if (lookahead == -1) {
        lookahead = cyr_lex();
        if(debug_flag > 1) {
                printf("TOKENS IDENTIFIED \n");
                printf ("Token for %s is %s\n",cur_lexeme,tok_strings[lookahead]);
                }
        }

/* Compare token with lookahead. */
return token == lookahead;

}
/******************************************************************
                s_term(char * pres_term)

Returns :       void

Purpose :       To collect a valid S_TERM.

Plan    :       Check if next term is S_TERM.
                If yes then copy into cur_lexeme.

Notes  :        None
******************************************************************/
static void s_term(char * pres_term)
{
/* Check if term is S_TERM */
```

```
if (match(S_TERM)) {
        /* Copy the term into cur_lexeme */
        strcpy(pres_term,cur_lexeme);

        /* Advance to next term. */
        advance();
        }
else
        printf("A valid term expected.\n");
}
/*****************************************************************
                where_term(char * wh1_term           First WHERE term
                        char * wh_comp_term          Comparison term
                        char * wh2_term              Second WHERE term


Returns :       void

Purpose :       To collect valid WHERE clause terms.

Plan    :       Do a series of match and advance corresponding
                to the grammar.
                First match wher1_term
                Next match wher_comp_term
                Next see if left brace
                        if yes then nested expression.
                else
                        match wher2_term

Notes   :       This is a recursive function if nested braces are
                found in WHERE clause.
*****************************************************************/

static void where_term(char *wher1_term,char *wher_comp_term,
                char *wher2_term)
{
if(match(WHERE_TERM)) { /* Match WHERE keyword */
        advance();
        s_term(wher1_term); /* Collect first where term*/
        if(match(COMP_TERM)){
                /* Collect comparison term */
                strcpy(wher_comp_term,cur_lexeme);
                advance();
                if(match(LFT_PAR_TERM)) { /* Match left brace */
                        advance();
                        expression();   /* Recursive call to expression */
                        if(match(RGT_PAR_TERM)) /* Match right brace */
                                advance();
                        else
                                printf("Mismatched parenthesis in WHERE clause.\n");
                        }
                else    {
                        s_term(wher2_term);/* Collect second where term */

                        /* Set no_errors flag */
                        no_errors = TRUE;
                        }
                }
        else
                printf("Comparison term expected in WHERE clause.\n");
        }
        else
                printf("WHERE clause expected.\n");
}

/********************************************************************
                from_term(char * fr_term)

Returns :       void

Purpose :       To collect a valid FROM term.

Plan    :       Check if next term is FROM.
                If yes then collect the term.

Notes   :       None
********************************************************************/
static void from_term(char *fr_term)
{
if(match(FROM_TERM)) { /* Match FROM keyword. */
        advance();
        s_term(fr_term); /* Collect the from term. */
        }
else
        printf("FROM clause expected.\n");
}

/***************************************************************
                select_term(char * pres_term)

Returns :       void

Purpose :       To collect a valid SELECT term.

Plan    :       Check if next term is SELECT.
                If yes then collect the term.

Notes  :        None
***************************************************************/
static void select_term(char sel_term[][32])
{
if(match(SELECT_TERM)) { /* Match SELECT keyword */
        advance();
        s_term(sel_term[sel_term_p]); /* Collect sterm */
        sel_term_p++;                   /* Increment sterm index */

        /* Collect all the select terms. Select terms are
         * separated by SEPARATOR like ",".
         */
        while (match(SEP_TERM)) { /* Match Separator */
                advance();
                s_term(sel_term[sel_term_p]);
                if(sel_term_p <  10)
                        sel_term_p++;
                else
                        {
                        printf("Internal error. More than 10 select terms.\n");
                        exit(1);
                        }
                }
        }
else
        printf("SELECT clause expected.\n");
```

```
}

/******************************************************************
                expression(void)

Returns :       void

Purpose :       Enforce the grammar.
                Collect the required terms.
                Call the translator.

Plan    :       Initialize all the variables.
                Collect SELECT FROM WHERE terms.
                Call translator to generate a derived class.

Notes   :       This is recursive function. All the variables are
                automatic because they are created on stack for
                each call to expression().
******************************************************************/
static void expression(void)
{
char sel_term[10][32],          /* Array to hold SELECT terms.                  */
     fr_term[32],               /* Array to hold FROM term.                     */
     wher1_term[32],            /* Variable to hold first WHERE term.           */
     wher2_term[32];            /* Variable to hold SECOND WHERE term.          */
char wher_comp_term[32];        /* Array to hold WHERE comparison terms         */
int i;                          /* Loop counter                                 */

/* Initialize the variables to NULL */
memset(fr_term,0,32);
memset(wher1_term,0,32);
memset(wher2_term,0,32);
memset(wher_comp_term,0,32);

for(i=0;i<10;i++)
        memset(sel_term[0],0,32);

/* Assume there is a error in expression .*/
no_errors = FALSE;
sel_term_p = 0;
```

```
/* Expression is of form SELECT FROM WHERE */
select_term(sel_term);
from_term(fr_term);
where_term(wher1_term,wher_comp_term,wher2_term);

/* If no errors then call translator functions */
if(no_errors) {
        if(debug_flag > 1) {
                printf("\nTERMS IDENTIFIED\n");
                printf("Select Terms = %s \n",sel_term[0]);
                printf("From Term = %s \n",fr_term);
                printf("Where Term1 = %s Comp_term is %s Term2 = %s\n",wher1_term,
                        wher_comp_term,wher2_term);
        }

        /* This set creates one derived class. */
        wr_class_name();
        wr_base_names(fr_term);
        wr_guard_rule(wher1_term,wher_comp_term,wher2_term);
        wr_methods(sel_term[0]);
        wr_end_class();
        }
}

/*****************************************************************
                s_term(void)

Returns :       void

Purpose :       To parse set of statements.

Plan    :       Call statements().

Notes   :       This function is redundant now. But may be needed
                in future.
*****************************************************************/
static void statements(void)
{
expression();
}
```

```
/*******************          Public functions          *******************/
/*****************************************************************
                    main(int argc, char ** argv)

Returns :           int

Purpose :           To collect the input.
                    Open and close the OUT_FILE.

Plan    :           Get the input. Convert input to uppercse.
                    Open OUT_FILE.
                    Call parser.
                    Close file.


Notes  :            None
*****************************************************************/
main(int argc, char ** argv)
{
if(argc > 1)
        debug_flag = atoi(argv[1]);

if(!gets(input_buffer)) {
        printf("Unable to get input.\n");
        exit(1);
        }
/* Convert input to uppercase */
strupr(input_buffer);

init_op_file();
statements();
close_op_file();

return 0;
}
```