A Translator from Object Structured Query Language into Object Algebra Expression

Sainaaz Bi Hussain

This thesis is submitted in partial fulfilment of the requirements for the Masters of Science Studies (Computer Science).

Department of Computer Science
University of Tasmania
Australia

January 1994

Declaration of Originality
To the best of my belief this thesis does not contain any material previously published by another person, except where the reference is made in the text of the thesis.

[illegible]
Sainaaz Bi Hussain

Acknowledgment

Many thanks to Dr. Chris Keen for his guidance throughout the project.

I would also like to thank Dr. Vishv Malhotra and Dr. Charles Lakos for their help in getting the parser working. Also many thanks to Carl Lewis for giving valuable ideas on the implementation of the parser, Sunee Pongpinigpinyo for helping in the selection of the Object Algebra, Crystal Austin and the Overseas Students Office for proofreading the thesis.

Last but not the least I would like to thank my husband(Yakub Hussain) and son (Wafi Isham Hussain) for their support and encouragement throughout the project.

Abstract

Object-Oriented database management systems have been proposed as the effective solution to providing the database management facilities for complex applications. This project involves in deriving a set of rules which specifies the translation of Object SQL statements into an Object Algebra expression.

The language chosen for this project is Object SQL (**Object** Structured Query Language). Object SQL is a high-level user language whose standard does not exist yet.

OA (Object **A**lgebra) is an intermediate level target language designed for a range of user languages. The algebra used in this project is defined by Dave Straube. This algebra is used as the target language for the translation of the Object SQL query statement into the equivalent Object Algebra expression. Object algebra is used for the procedural specification of queries which can then be used for the optimisation of queries.

Table of Contents

Chapter 1. Introduction

1.1. Aims

The aim of this project is to design a translator which can translate from Object Structured Query Language (Object SQL) into an Object Algebra (OA) expression. Therefore, one of the main contributions of this project is the derivation of a set of rules to specify the translation of a large subset of Object SQL statements into the equivalent Object Algebra expressions.

The major conceptual problem in this project is determining a scheme for compiling Object SQL constructs into Object Algebra expressions. The first part of the project is to select an Object Algebra. I will use the Object Algebra proposed by Dave Straube [Straube 1990a]. Then using this Algebra the translation from Object SQL into Object Algebra expressions will be performed. The categories of Object SQL statements are :

a) **DATA DEFINITION** Commands
For example : CREATE.

b) **DATA MANIPULATION** Commands
For example : SELECT, DELETE, UPDATE, and INSERT.

1.2. Definitions

Relational Database [Date 1990] : defines a relational database as a database that is perceived by its users as a collection of tables (and nothing else but tables).

Object-Oriented : a software development strategy that organises software as a collection of objects that contain both data structure and behaviour.

Object-Oriented Database Management System (OODBMS) : This is a database system which supports an object-oriented model. It manages complex objects with object identity, supports objects that encapsulate data and behaviour, structures objects in classes, and organises classes in a hierarchy. The OODBMSs augment the programming language by providing persistence, concurrent control, a query language, and other Database Management Systems (DBMS) capabilities [Catell 1992]. The application programmer may access database objects directly using the data-access operations in the programming language, or may perform associative lookups of objects using the query language.

Query : this is a language expression that describes what data is to be retrieved from a database and initiates the retrieval operation in the Database Management System.

Query Optimisation means strategies which seek to improve the efficiency of query evaluation so that there is minimum use of expensive resources such as the time involved in using the central processing unit (CPU), the secondary storage cost, and the storage cost.

Query Languages : this is a special purpose programming language used in the database system to formulate queries.

Object represents entities and concepts from the application domain being modelled; a concept, abstraction or thing with definite boundaries and meanings for the current problem; an instance of a class or type; or encapsulation of data and functionality (behaviour), for example, a car, a cup, a person. Objects can be classified as

- **literal objects** : these are self identifying objects such as integers and character strings.
- **surrogate objects** : these are objects that are represented by system generated, unique, immutable object identifiers or OIDs.

Type : a function type is to characterise objects in terms of the functions that can be applied to the object.

Class : a description of a group of objects with similar properties, common behaviour, common relationships, and common semantics.

Function denotes attributes of objects, relationships among objects, and operations on objects.

Variables : objects can be assigned to variables which can be used as temporary placeholders for results produced by function calls.

Extension : the extension of a function defines the mapping between the arguments and the results of that function. The extension of a type is the set of objects of that type.

Instances : objects in the extension of a type are the instances of that type.

Abstraction consists of focusing on the essential, inherent aspects of an entity while ignoring its accidental properties, that is, focusing on what an object is and does before deciding how it should be implemented.

Encapsulation means hiding representation and implementation in order to enforce separation between the external interface of an object and its internal implementation.

Inheritance : an Object-Oriented mechanism that permits classes to share attributes and operations based on a relationship; deriving new definitions from existing ones; the way in which classes relate to each other.

Identity : the characteristic of an object that provides a means of uniquely denoting or referring to the object independent of its state or behaviour.

Abstract Data Types : this is a class of data structures described by an external view.

Late Binding : an object may be created and stored under the version n of its class definitions.

Schema : a collection of definitions of types, classes and objects.

Recovery : The process of reproducing a consistent state of a system after a failure. The failure may be in an application process, the OODBMS, the operating system, or the underlying hardware.

Complex Objects are the objects which are built from other data structures or other objects.

Polymorphism : the property that an operation may behave differently on different classes.

Data Integrity relates to unauthorised access.

Extensibility : in the Object-Oriented systems it is easier and safer to add extra program components.

Code Reuse : this means using the existing code in different programs. Reusability of code (by specialising a class via inheritance) is a very powerful, robust, and safe way of extending the existing modular codes (that is, creating a new class just like the parent class but with some differences).

Ease of Use : in an Object-Oriented environment ease of use comes with experience.

Views : this can be thought of as different ways of looking at "real" objects.

Multi-User Concurrency Control : a mechanism that controls simultaneous sharing of objects among processes.

Predicate : this is a Boolean expression that can be established as TRUE or FALSE for a given record by examining that record in isolation.

Procedural Language : A language is referred to as procedural if the user has to specify in detail the steps necessary to obtain the information he wants to retrieve.

Data Definition Language (DDL) : This is a traditional term for the subset of the constructs of a database language in which the data model is declaratively expressed.

Data Manipulation Language (DML) : This is a traditional term for the subset of the constructs of a database language in which the behaviour is specified.

1.3. What is Original?

Designing and implementing the Object SQL parser was the first part of the project. Since this project was a continuation of a 1992 honours project, it was assumed at the beginning of the project that a parser had already been written.

Initially the parser did not compile but after working on it for a few weeks, I managed to compile it only to find that a lot of files were either missing or not written at all. Also the parser was written in C++ and the E (extension of C++ language) language about which I have very little knowledge.

I decided to write another Object SQL parser. Leroy Cain [Cain 1989] wrote an ANSI SQL parser in 1989. Using the information (on Object SQL) supplied by Hewlett-Packard [Lyngbaek 1991], this grammar was modified so that it was similar to Object SQL. Most of the operators used in the translation rules are those defined by D. Straube in his Ph.D thesis [Straube 1990a].

1.4. Brief Literature Review

Object-Oriented database systems have been proposed as an effective solution for providing data management facilities for complex applications [Straube 1990a].

In addition, they combine the advantages of Object-Oriented Programming and semantics of data modelling (abstract data types, encapsulation, inheritance, polymorphism, extensibility) with the advantages of traditional data management systems (declarative queries, set-oriented access, views, access control, and multi-user concurrency control) [Lyngbaek 1991].

Object SQL is a database language designed for Object-Oriented database systems. It supports user-defined operations in addition to the traditional data definitions and manipulations [Lyngbaek 1991].

Queries are an important component of database systems as query languages define the user interface (both syntactically and functionally) and the query processing techniques affect performance [Straube 1990a]. Many query languages have been developed for databases. [Straube 1990a] describes languages such as SQL and Object SQL as "user" languages whereas languages such as calculus and algebra are considered as "formal" languages. A calculus allows queries to be specified declaratively without any concern for processing details. On the other hand queries expressed in an algebra are procedural in nature, and this can be optimised. Algebra provides a sound foundation for rule-based transformation systems which allow experimentation with various optimisation strategies [Straube 1990a].

1.5. Chapter Summaries

In Chapter 2 a brief background to the theory of query languages, Object SQL and Query Algebra is outlined. This chapter also explains why Straube's Object Algebra was selected for this project.

The experimental design of the project is presented in Chapter 3. The YACC/LEX grammar and the translation rules are also presented in this chapter.

Chapter 4 presents the results.

The final chapter has some concluding remarks together with possible directions for future work.

Chapter 2. Background

Introduction

Indexed files were the earliest form of database management systems. These files provided a simple way of storing records and data. In the 1970s, the first complete database management system evolved. These used hierarchical and network models of data. Then in the 1980s relational databases such as INGRES and ORACLE were developed and commercialised. Since then relational database management systems have been widely used.

In the early 1980s Object-Oriented data modelling appeared. Since then, this field of Object-Oriented database has turned into a major research area and has become reasonably mature. The origin of Object-Oriented systems comes from the object-oriented programming environment where the main idea is that the user should be able to deal with objects and operations that closely resemble the real world. The main advantage of OODBMS over RDBMS is their query languages and rich data structures. In OODBMS the query languages support a high level of data abstraction because these database systems are based on object-oriented data models.

2.1. Object-Oriented Database Management Systems

Object-oriented database management systems have the advantage of both object-oriented programming and the semantics of data modelling along with the advantages of traditional data management systems. Its benefits to application developers include reduced schema complexity, an application code that is simpler to develop and easier to maintain, code reuse, extensibility, increased programmer productivity, and ease of use. This means more reliable and less costly applications.

Object-Oriented databases differ from relational databases in a number of ways. Object-Oriented databases allow reuse of code, and provide easy extensibility and increased productivity. Relational databases cannot manage complex data, while Object-Oriented databases can because they tightly integrate object programming languages with database support. Relational databases use separate languages and tools which create barriers between program design, coding, and database access. Crossing these barriers is very costly in terms of performance and productivity. Object-Oriented database programs are organised around their fundamental concepts, not along lines determined by the programming tools. Object-Oriented databases use a single language that handles in-memory data structures and database access, thereby markedly improving productivity.

According to [ONTOS 1991] some of the main features of Object-Oriented databases are :

1) **Persistence** : This feature saves objects on the disk so that they can be used in future sessions and can be shared by many users. Without persistence, objects would disappear when the program is completed.
2) **Data independence** : Application programs have abstract interface to data, which hides the data structure inside the database from the application. The database can be changed or recognised without changing any application code, greatly improving productivity.
3) **Query** : A query is a filtering or sorting mechanism. A collection of records can be compared with the query and those that satisfy the query are returned. A query is an excellent mechanism for finding objects in a database based on common characteristics.
4) **Transaction** : A transaction is a single unit of work performed by an application. It ensures that changes to the database are made in a consistent fashion. Either the transaction succeeds and all of the changes are committed to the database, or the transaction is aborted and none of the changes are made. Transactions insure that the database remains in a consistent state.
5) **Concurrency** : Concurrency is the ability for multiple users or applications to share the same data in a controlled, predictable fashion. Applications need to share data, but transactions tend to isolate them from each other. Concurrency mechanisms work together with the transaction model to synchronise simultaneous accesses to information in the database. Transaction and concurrency combine to create a powerful and flexible set of tools for development in complex application areas which demand the utmost in speed for data storage and retrieval.

2.2. Query Languages in OODBMSs

Introduction

All database systems have some sort of protocol for accessing their information content. These protocols most often take the form of a programming language or its extension. However, a major problem encountered with a programming language as a database interface is that programmers are the only people capable of using the database. Hence, the query concept is used to overcome this problem, that is, it allows individuals with little or no programming expertise to effectively use a database.

A query is a single interaction between an end-user and a database, that is, it is used for specifying what data is to be retrieved, from where it is to be retrieved or which data is to be updated in the database. The user formulates a request in a non-procedural manner, specifying 'what' to do rather than 'how' to do it. In other words, a query does not specify how to manipulate the data. The query processor analyses the request, and dynamically constructs and executes a program to respond to the request. Queries usually operate on sets or collections of instances and may return or affect a single or a collection of objects.

A query language is a frame for expressing a query. It consists of data definition and data manipulation languages.

Traditionally the database query languages had only minimal type checking requirements. For example, in the relational model type checking ensures that relational schemes are compatible and that only appropriate comparison operations are performed on the tuple fields. This is because this model supports only a limited number of primitive domains (for example : integer, string, boolean). However, object-oriented database query language introduces complexity into this process since query results may be homogeneous sets of objects, that is, all objects in the query results are not of the same type.

According to [Cattell 1992] an OODBMS should satisfy the following goals for query languages :

- It must provide a declarative DML that can be used for ad hoc operations.
- A high degree of physical data independence must be provided by the query language processor.
- An OODBMS must support a programming language that includes query language as a subset.
- The DML should allow access to all data structures, and provide at least the capability of the relational calculus with collection results.

2.2.1. Object SQL [Lynbaek 1990]

SQL (Structured Query Language) is a well-known and widely used query language for relational databases. It was originally proposed by IBM in 1981 as the user interface language to the System R relational database [Harris 1990]. This product was released in 1986. The first commercial SQL product developed was ORACLE. Since that time, several other implementations have been made. Because of the general acceptance of SQL in the 1980s, ANSI undertook languages standardisation.

Since the object paradigm is gaining attention in the programming community, and because SQL already enjoys considerable popularity, we can expect to see various implementations of an *Object SQL* from relational and object system vendors.

Object SQL is a subset of the ANSI SQL standard with object processing extensions. It is a language for posing questions about objects using the SQL paradigm. It is a high-level database language for object-oriented database management systems that supports both user-defined operations and traditional data definition and manipulation languages. It was defined to create a standard language for object-oriented databases that would be independent of any specific application programming language and specific implementation of object-oriented database management systems. Its main aim is to provide a database interface with modelling constructs which closely matches the real-life situations and the needs of business or technical applications. It has the advantages of both object-oriented programming : abstraction, encapsulation, extensibility and reuse, and the relational data definition and manipulation languages : declarative queries, set-oriented access, views, and authorisation. Being a functional language with special syntactic form, Object SQL closely resembles SQL for common database functions. The four basic constructs of Object SQL are :

1) **Objects** - The two types of objects are :
   - *literal objects* : They may be
     - **atomic,** such as Num, Char, Binary, Boolean, Date, Time, DateTime, Decimal, Float, Integer, LongInt, SmallInt.
     - **aggregate,** such as Set, Bag, List, Tuple. A set or bag is an unordered collection of objects whereas a tuple is an ordered, fixed-sized collection of objects.
   - *surrogate objects.*

2) **Type** - Types are related in a subtype/supertype hierarchy, known as the **type hierarchy**. The type hierarchy models the mode type containment, that is, if an object is an instance of a type it is also an instance of all the type's supertypes. In Object SQL a given type can have multiple supertypes. Object SQL's aggregate types (sets, bags, lists, tuples) support arbitrarily complex objects and can be composed of other types and objects using a set of object constructors. The constructor [. . . .] constructs bags, { . . . . } constructs sets, [| . . . . |] constructs lists, and < . . . . > constructs tuples. For example, {char} is a set of character strings and { 'green', 'red', 'blue' } is an instance of the above type.

3) **Functions** - Functions that perform database updates by updating other functions may change the state of the database as a side-effect of their execution. A function consists of a declaration (defining the signature of the function and the constraints of its extension) and implementation (defining the behaviour of the function). A stored function maintains its extension as an associated, persistent data structure and so has no side-effects on the database state. A function whose body is defined by an Object SQL statement may query or update the database. If the defining Object SQL program is a query, the function is called a **derived function** (has no side-effects) whereas if the defining program contains a statement other than a query, the function is called a **procedure** (has side-effects).

Object SQL is not tied to a specific language but is intended to combine the best from object-oriented programming languages with the benefits of declarative data definition and manipulation languages of SQL. Specifically, Object SQL was designed to provide the following goals :

- Provide object-oriented features such as improved productivity, extensibility, and code reuse.
- Provide declarative database language features such as association access, query optimisation, alternative views of data and integrity.
- Since Object SQL is a semantic superset of SQL, it provides the current technology features of SQL such as multi-user access, and authorisation/security.
- Adopt a functional language approach, since the fundamental Object SQL primitive is function invocation. Hence, features and functionality of the language are provided by a set of built-in functions; for example, the **TypeCreate** function creates new types and **FunUpdate** function updates the database.

- Ensure that the Object SQL syntactic form and keywords closely resemble SQL wherever possible in order to lessen the learning curve for Object SQL programmers.
- For built-in functions, it should provide language constructs for common data definition and manipulation tasks, in the form of Object SQL statements. For example, the *SELECT* statement is a syntactic embodiment of the *SELECT* function.

4) **Variables** - Since Object SQL supports variables, they can be used to communicate data between Object SQL and an application program. There are two types of variables :
   1) **Local variables** - these have a declared type. They can be declared inside function bodies and in the FOR EACH clause of certain Object SQL statements. They have limited scope, that is, the scope of a variable declared in a function is limited to the body of that function and the scope of a variable declared in a given statement is limited to that statement.
   2) **Session variables** - are global, do not require explicit declarations, cannot be used inside function bodies, can be thought of as being implicitly declared of type Object, and can be used any time during an Object SQL session. An object of any type can be assigned to a session variable.

2.2.1.1. Object SQL Statements

Object SQL statements are provided for defining, implementing, and declaring types and functions. The definition of types and functions in an Object SQL application correspond roughly to the definition of tables and views in an SQL application.

a) Data Definition

i) Defining Types

CREATE TYPE - defines a new type by specifying its name and its supertypes, for example, the *TeachingAssistant* shown in *Figure 2.1* below can be created by the following statement.

```
CREATE TYPE TeachingAssistant SUBTYPE OF Student, Teacher;
```

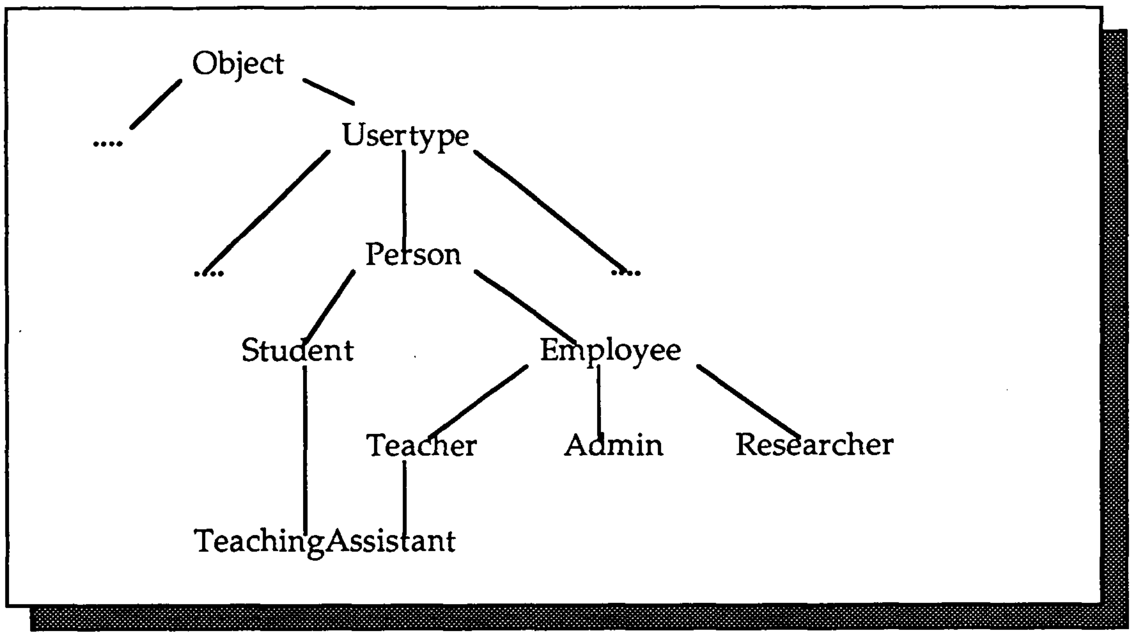

Figure 2.1 [Lynbaek 1991]

If the *SUBTYPE* clause is omitted then the new type automatically becomes a subtype of the built-in type *UserType*.

A type can also be created together with a set of functions on the new type, for example, the *Employee* and its functions shown in *Figure 2.2* below can be created with the following statement :

```
CREATE TYPE Employee SUBTYPE OF Person FUNCTIONs (
        FixedSalary Integer,
        Salary Integer AS FORWARD,
        DateOfHire Date,
        SocSecNum Char[11] UNIQUE
);
```

```
Employee subtype of Person
   FixedSalary
   Salary = FixedSalary
   DateOfHire
   SocSecNum

Teacher subtype of Employee

Admin subtype of Employee
   OvertimePay
   Salary = FixedSalary + OvertimePay

Researcher subtype of Employee
   ContractPay
   Salary = FixedSalary + 1/2 * ContractPay

TeachingAssistant subtype of Student, Teacher
   UnitsTaught
   Salary = UnitsTaught * UnitPay
```

Figure 2.2 : Payroll Application [Lynbaek 1991]

The optional keywords *AS FORWARD* specified for the function *Salary* shows that implementation is deferred although the function is being declared. A function can not be used until its implementation is specified. Those functions without an *AS FORWARD* specification are implemented as stored functions.

The optional keywords *UNIQUE* specified for function *SocSecNum* indicates a uniqueness constraint; that is, each result of the function is a unique string. Other optional keywords such as *DISJOINT* specified for a multi-valued function constraint indicates that the bag and set result objects of the functions are disjoint, that is, a given object can belong to at most one of the result objects of the function.

ii) **Defining and Implementing Functions**

a) CREATE FUNCTION - This statement defines a new function, for example, the function *Marriages* is defined as follows :

```
CREATE FUNCTION Marriages(Person) -> {<Person, Date>} AS
FORWARD;
```

The *AS* clause specifies how the function is implemented and *FORWARD* clause defers the implementation. The keywords AS STORED specifies the stored function.

SELECT statement is used to implement a derived function; for example,

```
CREATE FUNCTION Salary(Employee e) -> Integer s AS
    SELECT s
    WHERE FixedSalary(e) = s;
```

defines a derived function, *Salary,* in terms of the function *FixedSalary*. Here the salary of the employee is defined to be equal to the employee's fixed salary.

The value of a specified function can be changed by using the *UPDATE* statement; for example,

```
CREATE FUNCTION RaiseAllSalaries (Integer incr) AS
    UPDATE FixedSalary(e) = newsal
    FOR EACH Employee e, Integer newsal
    WHERE newsal = FixedSalary(e) + incr;
```

defines the procedure, *RaiseAllSalaries*. The procedure body updates the function *FixedSalary* by increasing the salaries of all the employees by a certain amount.

b) IMPLEMENT FUNCTION - This statement provides an implementation for a previously declared function. The various options for implementations are the same as those for *CREATE FUNCTION* statements. For example, the function *Salary* declared on the type *Employee* could be implemented using the following statement :

```
IMPLEMENT FUNCTION Salary(Employee e) -> Integer s AS
   SELECT s
   WHERE FixedSalary(e) = s;
```

iii) Deleting Types and Functions

A type or function can be removed from the database schema if it is no longer needed. For example, the function *Salary* defined on the type *Employee* can be deleted by the following statement :

```
DELETE FUNCTION Salary.Employee;
```

The *ALL* option can be used to delete all the functions with a given generic name; for example :

```
DELETE TYPE TeachingAssistant;
```

deletes the type *TeachingAssistant*. A *CASCADE* option can also be used to automatically delete the types, subtypes, and the functions defined on the deleted types. The CASCADE option should be used to delete a function that is used in the implementation of other functions.

DELETE TYPE statements can be used to delete a type which has no functions defined on it and no subtypes

b) Populating The Database

CREATE statements are used to create new objects. This statement optionally allows variables to be bound to the new object for future direct reference to the object; for example :

```
CREATE Person :mary, :alex, :sue;
```

creates three new *Person* objects in the object identifier form such that the objects can be referred to by the variables *:mary, :alex, and :sue.*

It is also possible to specify selected functions of those functions to be initialised. This is very similar to inserting tuples into tables in an SQL database. For example :

```
CREATE Person FUNCTIONS (Name, Address, Birthdate, Children)
    :george ('George Smith', 'Santa Cruz', DATE'1955-03-18', {:kevin}),
    :linda ('Linda Norton', 'Sunnyvale', DATE'1957-02-23', {:alex, :sue});
```

creates two additional *Person* objects and initialises their *Name, Address, Birthdate,* and the name of their *Children* functions. A single-valued as well as a many-valued function can be initialised in the same Object SQL statement.

An object can have several types. Initially it has the type it was created with and all supertypes of that type. However, at any time additional types can be added or existing types deleted. *ADD TYPE* statements can be used to add types to objects, for example, the following statement gives the additional type *Employee* to the *Person* object representing *George Smith* and denoted by the variable *:george* :

```
ADD TYPE Employee FUNCTIONS (FixedSalary, DateOfHire, SocSecNum) TO
:george (2000, DATE'1990-02-01', 218-34-3342');
```

REMOVE TYPE statements can be used to remove types from objects, for example :

```
REMOVE TYPE Employee FROM :george;
```

DELETE statements can be used to delete an object explicitly. This has the effect of removing the object from each of its types.

c) Updating the Database (Function Update)

UPDATE statements can be used to explicitly update the updatable functions. An update is used to change the value of the functions for future calls. When a function is updated, all functions derived from that function are also updated; for example :

```
UPDATE Address(:linda) := 'San Jose';
```

The value of function *Address* for object *:linda* is changed to *'San Jose'*. The "+=" assignment operator may extend the existing value of a function by adding components, while the "-=" assignment removes components from the existing value of a function.

d) Querying The Database

The *SELECT* statement provides the basic query facilities of Object SQL and has three parts :

SELECT *result specifications*
FOR EACH *local variable declarations*
WHERE *predicate*

- **Result specification** in the SELECT clause determines the objects to be returned. It may contain constants, functions (a single function, several functions separated by a comma, a list of functions, a tuple of functions, a set of functions, or a bag of functions), reserved functions (AVG, SUM, MIN, MAX, COUNT), expressions (consists of multiplication, division, addition or subtraction of constants, functions or reserved functions. For example, 5 * AVG(Salary(p)) or Salary(p) + OvertimePay(p)).

- **Declarations of variables** in the FOR EACH clause defines a set of target values for the variables.

- **Predicate** specified in the WHERE clause specifies a criteria for pruning the variables to a final set. It is a boolean expression consisting of function calls (including recursive SELECT Calls), literal objects, local and session variables, comparison (<, >, <>, =, <=, >=) and boolean operators (AND, OR, and NOT).

The SELECT statement always returns a Bag object. For example *Q2.1* returns a bag of character strings {[char]}. The WHERE clause is optional.

The following queries illustrate some of the different forms of the SELECT statement.

Example Q2.1 : Obtain the names of all people who live in San Jose.

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'San Jose';
```

Example Q2.2 : Obtain the names of all people who live in either San Jose or Los Angeles.

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'San Jose' OR Address(p) = 'Los Angeles';
```

Example Q2.3 : Obtain the names of all people whose age is greater than 40.

```
SELECT Name(p), Age(p)
FOR EACH Person p
WHERE Age(p) > 40;
```

Example Q2.4 : Obtain full details of all employees.

```
SELECT *
FOR EACH Employee e;
```

Example Q2.5 : Obtain the names of all people who have a son named ALEX. This is an example of a nested query where a SELECT statement appears within the predicate of another SELECT statement. Nested queries use IN, NOT IN, EXISTS, NOT EXISTS to specify whether something is present in another SELECT statement or not.

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) IN
      (SELECT Name(c)
      FOR EACH Children c
      WHERE Name(c) = 'Alex');
```

OR

```
SELECT Name(p)
FOR EACH Person p, Children c
WHERE Name(c) = 'Alex' AND Name(c) IN Children(p);
```

Complex Queries

The basic statements can be extended with the following clauses.

1) DISTINCT - specifies that the result should not contain any duplicates.
2) UNION operator - combines the results of two subqueries into a single result with no duplicates. The UNION ALL operator concatenates the results of two subqueries and has duplicates.
3) GROUP BY - groups objects together into bags of related objects. The HAVING clause specifies a predicate on each bag formed by the GROUP BY clause.
4) ORDER BY - useful for producing reports. It specifies an order (ascending, descending) in which the results are to be returned.

e) Cursors

Object SQL supports the concept of a cursor which is a mechanism that allows query results to be obtained one (or several) at a time.

OPEN CURSOR - associates a cursor with a given query; for example, in the following a cursor called *names* is created for the retrieval of *Person* names :

```
OPEN names FOR
SELECT Name(p)
FOR EACH Person p;
```

FETCH - allows one or more objects from the query results to be returned. NEXT clause can be used to return several results; for example, the following query can be used to obtain the next five objects of the query results :

```
FETCH names NEXT 5;
```

CLOSE - closes and deletes the cursor. For example, the following statement closes the cursor :

```
CLOSE names;
```

f) Control Flow

Some of the basic control flow Object SQL statements used for defining procedure bodies are :

BEGIN ....END. This block contains an optional list of local variables followed by one or more statements. Here the scope of the variable is limited to the BEGIN ...... END block.

IF ***predicate*** **THEN** ***statement*** **ELSE** ***statement.*** If the predicate is true than the first statement is executed, otherwise the second one is executed. The ELSE clause is optional.

WHILE ***predicate*** **DO** ***statement.*** The statement is executed if the specified predicate evaluates to true, otherwise the loop terminates. These steps are executed repeatedly until the predicate evaluates to false.

FOR EACH ***varname*** **IN** ***bag_specification*** **DO** ***statement.*** The specified statement will be executed for each set of binding (defined by the bag of specifications) of the specified variable.

g) Authorisation And Access Control

All database languages support access control and security property. The traditional SQL language provides support for controlled access to data. On the other hand, Object SQL which is based on a functional approach provides support for controlling the calling and updating of functions.

Object SQL provides statements for granting and revoking privileges to individual users and groups. The GRANT statement can be used to grant privilege. For example, the following statement grants call authority on the function *Salary.Employee* to the *salary-user* user :

```
GRANT CALL ON FUNCTION Salary.Employee TO salary-user;
```

The REVOKE statement can be used to revoke privileges. For example, the following statement revokes update authority from the *PUBLIC* group :

```
REVOKE UPDATE ON FUNCTION FixedSalary FROM PUBLIC;
```

Object SQL also provides statements for creating and deleting users, and changing their password.

h) Session And Transaction Control

Object SQL provides statements for database transactions and session control. These statements are very similar to SQL statements.

A transaction is a unit of work consisting of a series of accesses to the database. A transaction is started with the BEGIN WORK statement. The COMMIT WORK ends the current transaction and so all changes made during the transaction becomes permanent. The ROLLBACK WORK statement can also be used to terminate the current transaction.

A session consists of zero or more transactions and can be established between an application process and a database with the CONNECT statement, for example :

```
CONNECT TO testdatabase;
```

This statement also supports additional options for different system configurations, for example, the host machine name, login account name and a database password can be specified.

DISCONNECT statement is used to terminate a session, for example :

```
DISCONNECT testdatabase;
```

2.2.1.2. OODBMS for Object SQL query languages

1) ONTOS

ONTOS [ONTOS 1991], a product of Ontologic Inc., is a commercial Object-Oriented database based on the C++ language. It provides a persistent object store for C++ programs. It is based on a client/server architecture where multiple clients on different workstations can concurrently access object stored on one or more servers.

The schema of database is managed by DataBase (DB) Designer which is a browsing tool and an interactive visual schema design.

The queries can be stored as objects in a database. Thus queries which are used very often do not have to be specified repetitively. Once a query object is created it can be named and evaluated. Object SQL defines types for queries and query results. Users may freely define subtypes and supertypes so as to replace, refine or redefine the query processing operations.

2) OpenODB

OpenODB, developed by the Hewlett-Packard Company, is a hybrid OODBMS evolved from and coexisting with RDBMS. It includes an Object SQL language interface and a graphical browser on the client side and an object manager with a relational data storage engine and an external interface on the server side.

The object manager supports all Object-Oriented features; that is, it supports complex objects, dynamic schema modifications, dynamic typing, multiple inheritance, encapsulation, late binding, object identity, overloaded functions, a type hierarchy, and unlimited user-defined types.

The relational database, ALLBase/SQL, provides the data storage and manipulation capabilities such as authorisation, declarative queries, high availability, multimedia (for example, graphics, images, voice) support, multi-user concurrency support, recovery, referential integrity, and transaction management.

Object SQL can be used to create types, functions, and objects. Functions define attributes, and relationships that are retrieved or calculated with Object SQL statements.

This Object SQL also includes programming flow statements, such as IF....THEN....ELSE, FOR, and WHILE. This procedural language allows the definition of complex functions.

2.2.2. Query Algebra

Introduction

Currently, query algebra is gaining a lot of attention in the database area. The two main reasons for this are :

1) Query algebra provides an abstract language which provides not only the meaning of the queries but also the expressiveness of user query languages.
2) One of the important uses of query algebra is query optimisation.

A number of researchers are currently trying to define a query model and an Object-Oriented algebra which corresponds to the relational algebra used for the optimisation of the queries in relational databases. The two algebras considered for this project were the Encore query algebra [Shaw 1989] and the object algebra by Straube [Straube 1990a].

2.2.2.1. The Encore Query Algebra

In 1989 Shaw and Zdonik [Shaw 1989] developed an algebra for the Encore Object-Oriented database system. This algebra characterises all types as abstract data types whose implementations are hidden from the algebra.

This query algebra provides type specific operation against collections of complex objects with unique identities. It provides only two parameterized types, namely Tuples and Sets. The data model used in this algebra views everything as an object with an identity.

Most operations of this query algebra return collections of existing database objects as well as create new objects to store the requested relationships. The operators preserve the typing of the object-oriented data model.

Definitions of Encore Query Operators
This algebra defines the following operators :

1) **Select**. The operation creates a collection of database objects which satisfy a selection predicate. The operation is defined as :

$$Select\ (S,p) = \{s \mid (s \text{ in } S) \wedge p\ (s)\}$$

where S is a collection of objects and p is a predicate defined over the type of the objects in S. This operation creates a new collection object containing the identifiers of all members of collection S satisfying the predicate.

2) **Image and Project**. These operations are used primarily to return components of objects in the collection being queried.

The *Image* operation is used to return a single component or value for each object in the queried collection and has the following form :

$$Image\ (S, f : T) = \{f(s) \mid s \text{ in } S\}$$

where S is a collection of objects and f returns an object of type T.

The *Project* operation extends Image by returning multiple components of an object. Thus the maintenance of selected relationships between components of an object is supported. The relationships are stored as tuples, with the tuple type defined by the operation as follows :

$$Project\ (S, < (A_1, f_1), \ldots, (A_n, f_n) >) = \{< A_1 : f_1(s), \ldots, A_n : f_n(s) > \mid s \text{ in } S\}$$

where S is of type *Set[T]*, the A_i*'s* are unique attribute names, and each f_i takes a single input of type T and returns an object of type T_i.

3) **Ojoin.** This operation is an explicit join operator used to create relationships between objects from two classes in the database. It creates new tuples with unique identities in the database to store the generated relationships. Ojoin is modified to handle sets of objects which may not be tuples.

4) **Set operations.** The algebra includes set operations *Union, Difference,* and *Intersection* which are used to create new collections of objects. They are the general set operations with set membership based on object identity.

The other set operations are *Flatten, Nest* and *UnNest* which are used to restructure collections of objects.

The *Flatten* operation is used to restructure sets of sets and is defined in the following way :

$$\text{Flatten (S)} = \{r \mid \exists\, t\ t \text{ in } S \wedge r \text{ in } t\}$$

Nest and *UnNest* operations allow the representation of tuples as flat or nested relations. The *Nest* operation compares attribute values using object identity and collects values of objects into a set type. The *UnNest* operation converts a set type into a simple type.

A result collection may be either existing database objects or new tuple objects created during the operation. To maintain the identity of objects in the database, the algebra contains two identity operators :

DupEliminate (S, i) : This operator keeps only one copy of i-equal objects from a collection of objects.

Coalesce (S, A_k, i) : used for the collection of tuple objects. This operator eliminates i-equal duplication in the A_k components (attributes) of the tuples.

2.2.2.2. The Straube Object Algebra

This object algebra is a collection of operators on objects. Operands and results in the object algebra are set of objects. Hence the algebra maintains the closure property (the result of a query can be used as the input to another). Some of the operators are qualified by a predicate.

Definitions of Object Algebra Operators

This algebra defines the following six operators :

1) **Union** (denoted $P \cup Q$). Here the set of objects are in either P or in Q or in both.

2) **Difference** (denoted $P - Q$). Here the set of objects is only in P and not in Q.

3) **Intersection** (denoted $P \cap Q$) can be derived by P-(P-Q). Here the set of objects is in both P and Q.

4) **Select** (denoted $P\ \sigma_F <Q_1...Q_k>$). Select returns the objects denoted by *p* in each vector $<p,q_1,...,q_k> \in P \times Q_1 \times ....Q_k$ which satisfies the predicate *F*.

   For example, find all documents about cars by persons over 50 years of ages. Let *d* range over *Doc* and *p* range over *Person*. Then

   $Doc\sigma$ ["car" $\in$ <d>.keywords $\wedge$ p == <d>.author $\wedge$ "50" = x $\wedge$ "True" = <p,x>.age.greater] <Person>

   The result of this expression is a set of *Document* objects and not sets of <Document, Person> objects. This is because the algebra has an 'object preserving' nature and does not create new objects. Hence, **Select** is like a semi-join operator.

5) **Generate** (denoted $Q_1\gamma^t_F<Q_2...Q_k>$). F is a predicate with the condition that it must contain one or more generating atoms for the target variable *t* where *t* does not range over any of the argument sets. This operation is objects denoted by *t* in F for each vector $<q_1,...,q_k> \in Q_1 \times ....\times Q_k$ such that it satisfies the predicate F.

   For example, return all coauthors of the document "Object oriented concepts". Let t be the target variable and d range over Doc. Then

   Doc γ^t["Object oriented concepts" = <d>.title $\wedge$ t $\in$ <d>.coauthors]<>

The most common uses of **Generate** operators are :

- to collect results of method applications.
- to iterate over the content of set valued objects.

6) **Map** (denoted $Q_1 \rightarrow_{mlist} <Q_2...Q_k>$). Let *mlist* be a list of method names of the form $m_1...m_m$. Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in $<Q_2...Q_k>$ as parameters to the methods in *mlist*. This returns the set of objects resulting from each sequence application. If no method in *mlist* requires parameters, then $<Q_2..Q_k>$ is the empty sequence <>. Map is a special case of the generate operator.

2.2.2.3. Properties of Query Algebra

[Yu 1991] identified several features that could be used for evaluating query algebras. They are as follows : *object-orientedness, expressiveness, formal properties, performance,* and *database issues.*

a) **Object-Orientedness** : The four main properties that should be supported by query algebra are :

1) *Object identities* : To support object identity, an object algebra should define its semantics over identities. That is, its operations should take object identities as input and produce an object identity as output. Straube's object algebra is defined on sets of identities, supporting identity-test whereas Encore's query algebra is defined in terms of the identities of collection of objects.

2) *Encapsulation* : To support encapsulation, the definition of an object algebra should be restricted to the operation on objects only through their interfaces. Straube's object algebra operates on objects whose internal representation is inaccessible, while Encore's query algebra supports encapsulation by defining Tuple and Set objects.

3 *Inheritance hierarchy* : To support this concept, queries expressed in an object algebra should be allowed to be directed against an inheritance hierarchy root. Straube's object algebra uses the semantics of the inheritance hierarchy as a generalisation pattern. The leaves of their

query trees can be either a class name C or a class sub-hierarchy C* which includes all members of class C and its subclasses. On the other hand, Encore's query algebra supports type inheritance for integrity.

4) *Heterogenous sets* : Heterogenous sets are those sets which can have members of different types. They may have to be used to support the generalisation concept. For example, if a set has member type Person then it may contain objects of Person, Student and Employee. Both Straube's object algebra and Encore query algebra support heterogenous sets.

b) Expressiveness.

1) *Extends relational algebra consistently* : For query algebra to be at least as powerful as relational algebra, it should be provided with the five operations that are defined in relational algebra : *Selection, Projection, Cartesian product, Union* and *Difference.* These operations work on sets of objects rather than on objects only. Straube's object algebra has five operations : *Select, Union, Difference, Generate* and *Map* which are similar to the relational algebra. However, Projection and Cartesian product are not included in this algebra. Encore's query algebra is seen as an extension of relational algebra, for example, Ojoin with appropriate predicate specifications can simulate the Cartesian product defined in relational algebra. The five operations of Encore's query algebra are : *Select, Image, Project, Ojoin* and *Set operations.*

c) Formal Properties. The 2 main properties are :

1) *A Formal semantics.* It is important for the query algebra to have a precise mathematical definition for each of its operations. The correctness of proposed query evaluation algorithms should be provided mathematically. Straube's object algebra has a concise, mathematical semantics whereas Encore's query algebra has only a partially formal semantics.

2) *A closed algebra.* Any algebra should specify the types of objects it supports and the allowable operations on objects of each defined type. All legal operations should be closed, so that no operation produces a result which lies outside of the scope of the algebra. Thus all its operations take as input and produce as output objects of a single type. Both Straube's and Encore object algebra satisfy this property. Operations

can produce atomic or aggregate objects. Its set operations do not accept atomic and aggregate objects as operands.

d) **Performance.**

1) *Provides optimisation strategies.* The main aspect of the query algebra is the implementation efficiency. Therefore, equivalence properties between the algebra operations should be studied and strategies for optimising algebraic expressions should be provided. Straube's object algebra defines semantic transformations for some of its operations. Equivalence-preserving rewrite rules that can be applied to object algebra expressions are also defined. Most of the conditions under which a transformation occurs reduces the execution cost of an expression. Encore's query algebra defines a form of equivalence as weak equivalence [Shaw 1989]. It is based only on database objects returned by a query. i-equivalence recognises the preservation of the relationship between database objects and structural equivalence recognises differences in identities in the structures of objects storing query results.

2) *Supports strong typing.* Every variable must have its type declared and it should be only assigned objects of its type or of a subtype of its type. This property determines whether an algebraic expression is type safe or not. This can help in preventing run-time failures and the performance can be greatly increased. Both Straube's object algebra and Encore query algebra operate upon entity types, class names and collections of objects.

e) **Database Issues.**

1) *Support both persistent and transient objects.* The query algebra should provide assignment operations that distinguish between persistent and transient objects. Persistently tagged objects are retained as the permanent object store while transiently tagged objects stay in volatile memory. Straube's object algebra deals with persistent objects while Encore query algebra does not deal with this issue.

2) *Equivalent object calculus.* It is desirable to have an object calculus whose expressive power is equivalent to that of the object algebra as a basis for an end-user language. Straube's object algebra defines a calculus for their object algebra and the equivalence of expressive power between the calculus and the algebra, while Encore's query algebra does not define a calculus.

Table 2.1 summarises the two query algebras considered in this project.

Framework	Query Algebra	
	Straube	Encore
Object-Orientedness		
Supports identities	+	+
Suports encapsulation	+	+
Supports inheritence hierarchy	+	-
Supports heterogeneous sets	+	+
Expressiveness		
Extends relational algebra consistenly	-	+
Formal Properties		
A formal semantics	+	+
A closed algebra	+	+
Performance		
Provides optimisation strategies	+	+
Supports strong typing	+	+
Database Issues		
Support for persistent & transient objects	-	-
Equivalent Object calculus	+	-

Table 2.1: Comparison of Query algebras

KEYS :
"+" : satisfies
"-" : does not satisfy

There are two main features that make the Object Algebra an important internal representation which can be used in the optimiser. Firstly, Object Algebra is a procedural language since an expression in Object Algebra gives a set of operations on sets and the order in which they are to be performed. Secondly, Object Algebra is an intermediate level language designed for a range of user languages.

It is very important for an object algebra to support encapsulation so that it is consistent with the concept of abstract data type and it has a concise semantic. Moreover, it should provide optimisation strategies.

Although neither Straube's object algebra nor Encore's query algebra satisfies all the properties, Straube's object algebra is considered superior because it deals with formal semantics and its mathematical definition for its operators are more efficient than the Encore's query algebra.

Chapter 3. Implementation of the translator

Introduction

The first part of the project was to design and implement the Object SQL parser. The implementation of the Object SQL parser is based on Leroy Cain's version of the ANSI SQL parser written in 1989.

The implementation was written in YACC and LEX, which are available on most machines. After writing the parser, the translation rules for the Object Algebra expressions were added to the parser. The rules were written in C language.

Section 3.1 briefly describes LEX programming while section 3.2 describes YACC programming. Section 3.4 gives an overview of the system developed. It also describes how the translation was done.

3.1. An overview of LEX programming

LEX (LEXical analysis program generator) is a software tool that allows the user to solve a wide class of problems drawn from text processing (can be used to check the spelling of words for errors), code enciphering (can be used to translate certain patterns of characters into others), compiler writing (can be used to determine what tokens, that is, smallest meaningful sequences of characters or reserved words, there are in the program to be compiled), and other areas.

LEX can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, etc.

A LEX specification consists of at most three sections :

1) Definitions : may contain #include, or abbreviations,
2) Rules : each rule consists of a specification of the pattern sought and the action(s) to take on finding it, and
3) User subroutines : an action code that is to be used for several rules can be written here and called when needed.

The sections for definitions and user subroutines are optional but if they are present then they must appear in the order indicated; that is definitions, followed by rules, followed by user subroutines.

The lexical analyser that LEX generates (not the file that stores it) has the name yylex(). *Figure 3.1* given below shows the creation and the use of a Lexical analyser with LEX.

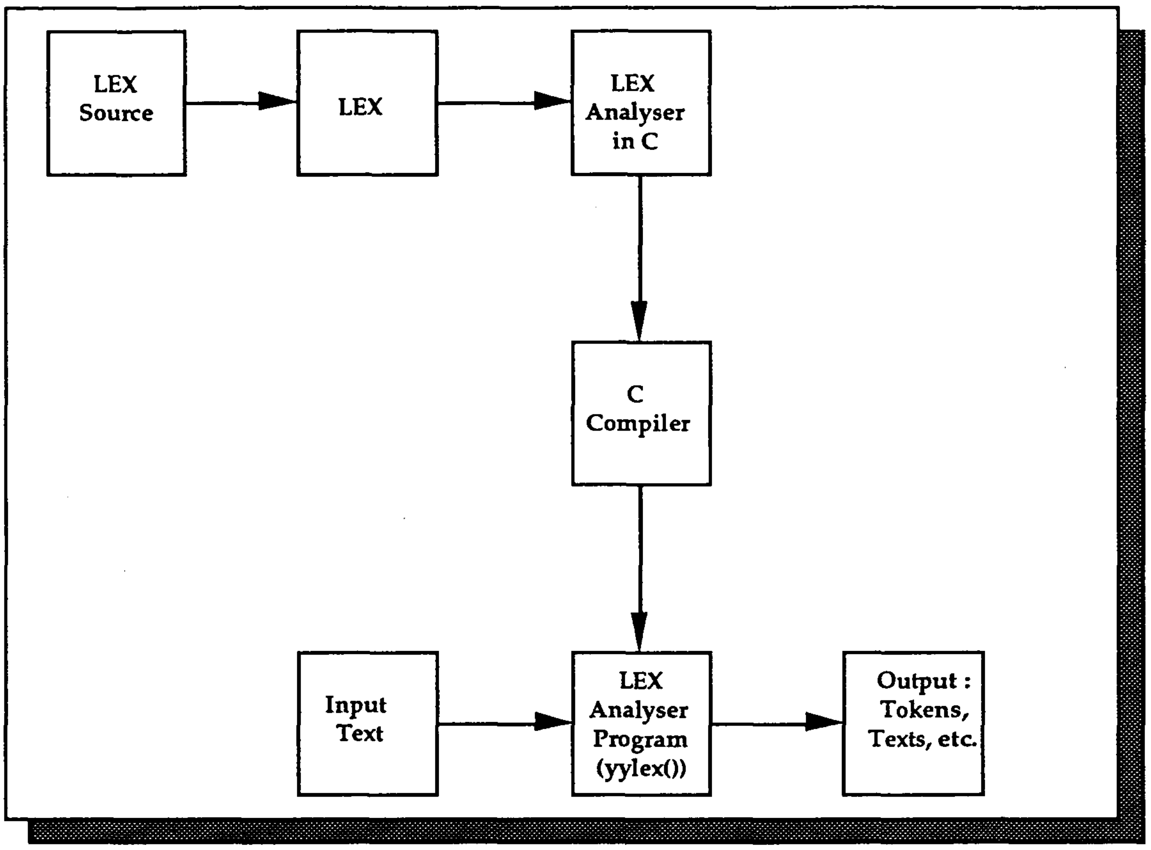

Figure 3.1 : An overview of LEX [Guide 1989]

3.2. An overview of YACC programming

YACC (Yet Another Compiler-Compiler) provides a general tool for imposing structure on the input to a computer program. The YACC user prepares a specification that includes :

- a set of rules to describe the elements of the input.
- the code to be invoked when a rule is recognised.
- either a definition or declaration of a low-level routine to examine the input.

YACC then turns the specification into a C language function that examines the input stream. This function (parser) works by calling the low-level input scanner (lexical analyser) which picks up the tokens from the input stream. Tokens are then compared with the input construct rules (grammar rules). When one of the rules is recognised the user code supplied (action) for this rule is invoked. Actions are fragments of C language code and so they can return values and also make use of the values returned by other actions.

Each grammar rule describes a *construct* and gives it a name. One grammar rule might be :

```
date : month_name day ',' year ;
```

where date, month_name, day, and year represent constructs; month_name, day, and year may be defined elsewhere. The comma enclosed in single quotes means that a comma will appear literally in the input.

The lexical analyser is an important part of the parsing function. This user-supplied routine reads the input stream, recognises the low-level constructs, and communicates these as tokens to the parser.

A full specification file looks like :

```
declarations {optional}
 %%
(grammar) rules
 %%
subroutines {optional}
```

YACC turns the specification file into a C language procedure, which parses the input according to the specifications given. The function produced by YACC is yyparser() which is an integer valued function.

3.3. YACC and LEX

YACC and LEX can be used on their own but often a combination is more appropriate. LEX is used to partition the input stream and the parser generator assigns structure to the resulting pieces. The generated program (by LEX), yylex(), is used by YACC for its analyser.

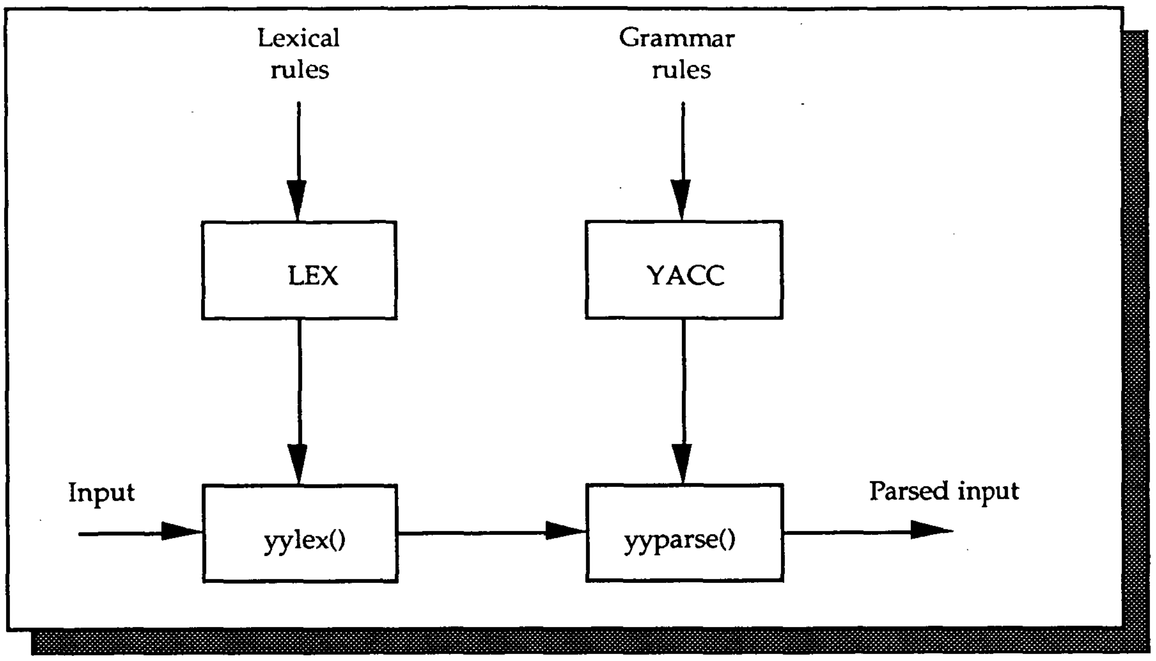

Figure 3.2 : LEX with YACC [Lesk]

3.4. System Overview

Figure 3.3 shows the design model of the translator from Object SQL into the Object Algebra expression.

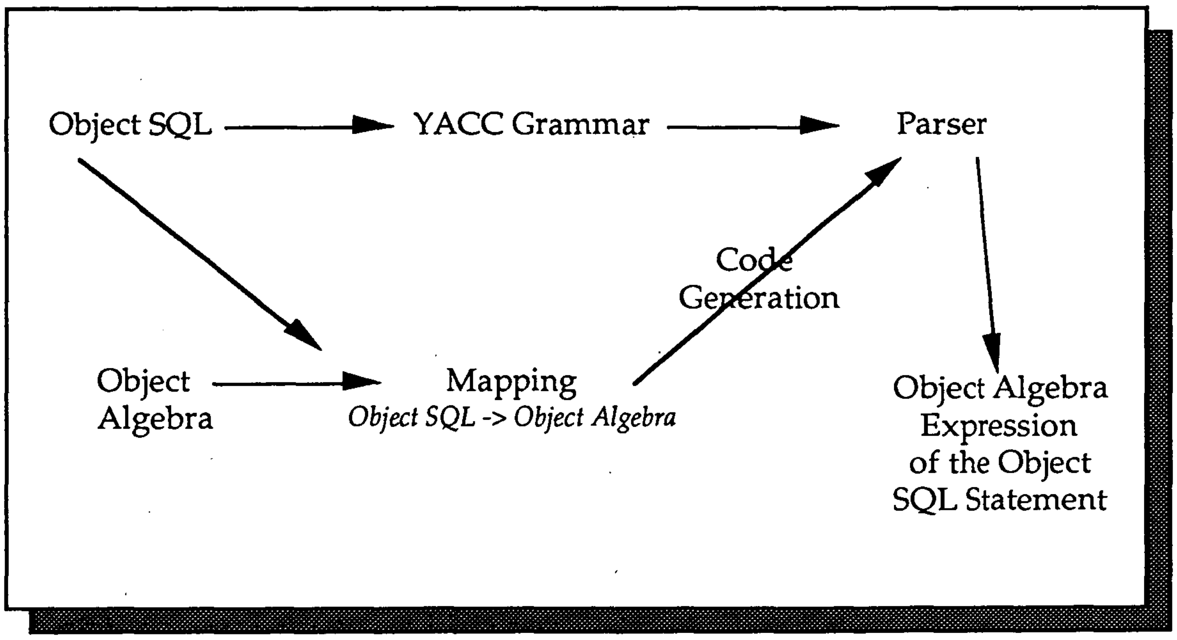
Object SQL
YACC Grammar
Parser
Code Generation
Object Algebra
Mapping
Object SQL -> Object Algebra
Object Algebra Expression of the Object SQL Statement

Figure 3.3 : Design Model of Object SQL Translator

Based on the design model shown above the following steps were undertaken to write the parser and the translator from Object SQL statements into the Object Algebra expressions :

1) Using YACC and LEX, Leroy Cain's ANSI SQL parser [Cain 1989], and the information supplied on Object SQL by Hewlett-Packard [Lynbaek 1991] the Object SQL parser was written.
2) The translation of Object SQL query statement into the equivalent Object Algebra expression was performed using Dave Straube's Object Algebra operators [Straube 1990a], namely UNION, INTERSECTION, DIFFERENCE, MAP, and GENERATE. Using the C language the parser was modified to write the translation rules.

The parser was written for all Object SQL statements, but the translation of an Object SQL statement into the equivalent Object Algebra expression was peformed only for the SELECT statement (query statement). Because of time limitations the translation rules for the GROUP BY, ORDER BY, and HAVING clauses for the SELECT statement were not written. Translation of the SELECT statement with only the session variable (SELECT Name(:Alex);) was also not done because the Object Algebra does not have the operator to handle this.

3.4.1. Modification of the Parser for the Translation Rules.

As each word of the Object SQL statement goes through the parser, it is recognised by the compiler as a token (that is, a reserved word such as SELECT, FOR EACH, WHERE, etc) or an identifier (that is, types, subtypes or supertypes, functions, predicate in the WHERE clause, etc).

i) Creation of data dictionary using the CREATE statement

Before parsing the query statement to produce the Object Algebra expression, the functions, types, subtypes, and supertypes have to be created and stored in the 2-dimensional array called *st_types* defined below. This is referred to as the ***data dictionary*** throughout the project.

```
st_types[MAXNUMIDS][MAXSTRING]
```

where

MAXNUMIDS :	this is the maximum number of functions, types, subtypes or supertypes that can be stored for translation of the Object SQL query.

MAXSTRING : this is the number of characters that each function or type can have. Function *Name* has 4 characters, for example.

The MAXNUMIDS and MAXSTRING values are defined in the header file named ***variables.h*** and can be changed if necessary.

The initialisation of all the arrays is done in the ***initilisation.c*** file.

The following file ***st_types.c*** shows how the types and functions are stored in the *st-types* array.

File st_types.c

```
/*****************************************************************************************************************/

include "variables.h"
#include <string.h>

store_types()
/* This procedure stores the types, subtypes, supertypes and functions in the data
dictionary so that it can be used in the translation of the queries. */
      {
      initialisationOfstoring_types();
      ml = 0;
      while ((ml < MAXNUMIDS) && (st_types[ml][0] != '\0'))
            ml++;
      if (ml == MAXNUMIDS)
            printf("ERROR in storing functions");
      else
            {
            strcpy(st_types[ml], s_type[0]);
            init_st_types = 1;
            }
      }

initialisationOfstoring_types()
/* This procedures initialises the st_types array only once during the execution of
the program so that more types can be added on.*/
      {
      if (init_st_types != 1)
            for (ml=0; ml<= MAXNUMIDS; ml++) st_types[ml][0] = '\0';
      }


/*****************************************************************************************************************/
```

Creation and Storing of types, subtypes, supertypes and functions can be done using the following four statements :

1) CREATE TYPE typename1, typename2, etc. Can have the MAXNUMIDS (defined in the variables.h header file) of types.

2) CREATE path_element FUNCTION (type_name_list) var_list. In this statement path_element is the name of the type, supertype or subtype and type_name (type_name_list) is used for storing functions.
3) CREATE object_name p_list. In this statement object_name is the name of the type, supertype or subtype.
4) CREATE path_element (t_list) r_type. Here also path_element is the name of the type, supertype or subtype.

More types, subtypes, supertypes and functions can be added to the data dictionary in each running of the program. That is, once the program is compiled and is running, a data dictionary consisting of the type, subtypes, supertypes and functions can be created. SELECT statements can then be parsed and the Object algebra expression of the statement obtained. At any time during the execution of the program new types, subtypes, supertypes and functions can be added to the data dictionary if the need arises.

ii) Storing of Result Specifications, Local variable declarations, and Predicates of the SELECT statement.

The basic form of the SELECT statement is as follows:

SELECT *result specification*
FOR EACH *local variable declaration*
WHERE *predicate*

The basic form of the Object algebra expression is as follows :

$Q1\ \gamma\ t\ F<Q_2 \ldots Q_k>$

where
Q1, Q2, ..., Qk are sets of objects, the symbol γ is the generate operator, **F** is the predicate, and **t** is the target variable.

i) Result Specification
As described in Chapter 2, the result specification determines objects to be returned. As applied to the Object Algebra expression, these result

specifications are the target variable (t) for the Object Algebra expression. The result specification of the parser constructed has the following forms :

a) One or more functions, types, subtypes or supertypes separated by comma or a tuple, list, bag, or set of functions. The functions can be single valued, for example, *Age*, *Name*, etc or multi-valued, for example, *Children* which may in turn have single-valued or multiple-valued functions.

b) Constants.

c) Reserved functions (Average, Minimum, Maximum, Count, and Sum).

d) Expressions with one of the above, that is, functions, constants or reserved functions and the manipulation operators : multiplication (*), addition (+), subtraction (-), or division (/).

NOTE : To avoid duplicates the word *DISTINCT* is used.

Some of the different types of expressions implemented are :

1) Simple expressions

a) SELECT DISTINCT Name(p).

b) SELECT DISTINCT Name(p), Age(p).

c) SELECT DISTINCT {| Name(p), Age(p) |}.

d) SELECT DISTINCT <| Name(p), Age(p) |>.

e) SELECT DISTINCT [| Name(p), Age(p) |].

f) SELECT DISTINCT [: Name(p), Age(p) :].

where
Name and *Age* are the functions and *p* is the range variable for the type. { |...| } is a set of functions, [: ... :] is a bag of functions, < | ... | > is a tuple of functions, and [|...|] is a list of functions.

In the above statements, *<p>. Name* and *<p>.Age* are the *target variables(t)* for the Object Algebra expression. When doing the translations for a list, bag, tuple, and set of objects the Object Algebra *MAP* operator is used.

The target variable(t) for the above simple expressions are :

a) $t \in$ <p>.Name

b)-f) $t \in$ <p>.(Name, Age).

Types, supertypes, or subtypes such as *Children(p)* which are multi-valued, that is, which have several single-valued functions can also be used in the result specification.

NOTE : A query may have only one of the above forms. Although the parser will not give an error message if more than one of the above result specifications is used in the query, the Object Algebra expression produced will not be correct. That is, if the result specification for a query has the following form :

b) SELECT DISTINCT [| Name(p), Age(p) |], < | Name(p), Age(p) | >

then the target variable (t) for the Object Algebra expression is t ∊ <p>.(Name, Age, Name(p), Age(p)). This is not correct because the specification needs a list and a tuple of functions.

Assumption : All the range variables in the set, list, bag, tuple, or result specification with more than one simple expression should be of the same type. That is, if the type is Person then the range variable should all be "p", range variable for type "Person" and not a combination of type Person, p, and subtype Children, c. For example, the following is correct :

SELECT DISTINCT [| Name(p), Age(p) |]

whereas this is not correct

SELECT DISTINCT [| Name(p), Age(c) |]

2) Expressions with constants

a) SELECT DISTINCT 5

where

5 is a constant. In the above statement, 5 is the *target variable(t)* for the Object Algebra expression; that is, t ∊ 5.

3) Expressions with reserved functions

a) SELECT DISTINCT AVG(Sal(p))

where

AVG (Average) is a predefined function, *Salary* is a function and *p* is the range variable for the type *Person*. In the above statement, *AVG(<p>. Salary)* is the *target variable(t)* for the Object Algebra expression, that is, t ∈ AVG(<p>.Salary).

Other types of predefined functions that were implemented include MIN (Minimum), MAX (Maximum), SUM, and COUNT.

4) Expressions with operators

a) SELECT DISTINCT [5 * Salary(p)]

where
5 is a constant, * is an operator for the expression, *Salary* is a function and *p* is the range variable. In the above statement, *(5 * <p>.Salary)* is the *target variables(t)* for the Object Algebra expression, that is, t ∈ (5 * <p>.Salary).

NOTE : A query can have any number of the above four types of expressions, that is, Simple expressions, Expressions with constants, reserved functions or with operators.

Each function, constant, reserved function, and the expression containing the operators that will be used for the translation into the Object Algebra expressions is recorded or saved in a 4-dimensional array; that is, an array with the following structure :

array name[...][...][...][...].

For example, if *t ∈ <p>.Name*, then function *Name* is stored in

function[MAXSELECT][MAXLEVEL][MAXNUMIDS][MAXSTRING]

where

MAXSELECT : this is the maximum number of subqueries or subselect statements. It is used for the translation of a UNION, INTERSECTION, MINUS, and DIVIDEBY of SELECT statements, for example, SELECT ......, etc. UNION SELECT .... MAXSELECT increases when there is a UNION, INTERSECTION, MINUS, OR DIVIDEBY in the query.

MAXLEVEL : this is the level of nesting in each SELECT clause, for example,

```
SELECT ......                         MAXLEVEL = 0
FOR EACH .....
WHERE ..... IN
              ( SELECT .....          MAXLEVEL = 1
              FOR EACH .....
              WHERE ... etc.)
```

MAXNUMIDS, and *MAXSTRING* have the same definitions as those defined for the *st_types* array.

The MAXSELECT and MAXLEVEL values are also defined in the header file named ***variables.h*** and can be changed if the need arises.

File st_function.c : The following four procedures show how the *function* array defined above is used for storing the functions.

```
/*****************************************************************************************************/
#include "variables.h"
#include <string.h>

check_func_or_types()
/* This procedure checks for the existence of the function present in the result
specification in the data dictionary*/
       {
       if (st_types[ml][0] != '\0')
              if (ml == MAXNUMIDS)
                     printf("Too many functions or types ");
              else
                     if (strcmp(function[0], st_types[ml]) == 0)
                            {
                            func_or_type = 1;
                            check_num_stats();
                            }
                     else
                            {
                            ml++;
                            check_func_or_types();
                            }
       else
              {
              printf("\n !!!! result specification");
              printf(" '%s' is not in the database", function[0]);
              }
       }

check_num_stats()
/* This procedure checks for the MAXSELECT level, that is, which subquery it is :
first, second, etc. */
       {
        int templ;
        templ = j + 1;
        if ((num_stats[j] == 1) && (num_stats[templ] == 0))
                {
                stat_level=j;
                store_functions();
                }
        else
                j++;
        }

store_functions()
/* This procedure checks for the MAXLEVEL, that is, the level of nesting in the
query. */
        {
         int temp;
        temp = i + 1;
        if ((level_nesting[stat_level][i] == 1) &&
        (level_nesting[stat_level][temp] == 0))
                {
                a_level=i;
                st_functions();
                }
        else
                i++;
        }

st_functions()
/* This procedure stores the function in the appropriate array */
        {
        f1 = 0;
        while((f1<MAXNUMIDS)&&(functions[stat_level][a_level][f1][0]!='\0'))
                f1++;
        if (f1 == MAXNUMIDS)
                printf("ERROR in function name");
        else
                strcpy(functions[stat_level][a_level][f1], function[0]);
        }
/*****************************************************************************************/
```

<u>Dissection of st_function.c file</u>

The procedures in this file check if the function, type, subtype or supertype is present in data dictionary. If it is present then it is stored in the appropriate array for translation.

Procedure *check_func_or_types()* checks whether the function to be selected is present in the database or not. The function is first stored in a two dimensional array called *function[..][..]*. This is done in the YACC specification of the parser. This function is then compared with the functions stored in the data dictionary. If the function or type used in the query is present in the existing data dictionary then the procedure *check_num_stats()* will be executed, otherwise the message *"result specification not in the data dictionary"* appears on the screen. Each time a new function is encountered when the query is parsed, the array function[..][..] is overwritten. (This is because the previous function has already been stored). If two functions are queried and only one of them is in the data dictionary then only the Object Algebra expression of the function in the data dictionary will be performed.

Procedure ***check_num_stats()*** checks the statement level; that is, whether it is a first subquery, second, third, etc.

Procedure ***store_functions()*** checks the level of nesting in the SELECT statement. The level of nesting increases if there is another SELECT statement in the WHERE clause of the previous SELECT statement.

Procedure ***st_functions()*** stores the function.

The constants (for example, 5), reserved functions (for example, AVG(Salary(p))), other expressions (for example, [5 * Salary(p)]), and range variables for types, subtypes, and supertypes are stored using a similar 4-dimensional array and a similar file as shown for storing the functions. *Table 3.1* summarises the arrays and files for storing the constants, range variables, reserved functions, and other expressions.

Expression and examples	Array name	File name
functions or types eg. Name(p), Children(p)	functions (eg. Name) ranges (eg. p)	st_functions.c st_funcranges.c
constants eg. 5	const1	constants.c
Reserved functions eg. AVG(Salary(p))	res_funcs (eg. AVG) col_funcs (eg. Salary) col_ranges (eg. p)	resfunction.c st_colfuncs.c st_colranges.c
Other expressions eg. [3 * Salary(p)] or [5 * MIN(Salary(p))]	other_funcs (eg. Salary) other_func_ranges (eg. p) other_consts (eg. 3, 5) other_reswords (eg. MIN) other_res_ranges (eg. p) other_resfuncs (eg. Salary) other_ops (eg. *)	st_otherfuncs.c st_otherranges.c st_otherconsts.c st_otherreswords.c st_otherresranges.c st_otherresfuncs.c st_otherops.c

Table 3.1 : Arrays and files for storing expressions

ii) Local variable declarations

The local variable declaration describes the set of target values for the result specification. These are either objects, types, subtypes, or supertypes. As applied to the Object Algebra expression, these local variable declarations are the sets of objects which are the arguments to the GENERATE operator, γ, that is, $Q_1 \, \gamma^t \, F <Q2, \dots, Q_k>$. Here Q_1 .. Q_k are the local variable declarations.

For example :

```
SELECT ....
FOR EACH Person p, Children c
```

where *Person* is an object, *p* is a range variable for *Person*, *Children* is a subtype of *Person*, and *c* is a range variable for *Children.* These are also stored in a 4-dimensional array and similar sort of file as shown above for storing the functions. *Table* 3.2 summarises the arrays and files for storing the types, subtypes, supertypes and the range variables for types, subtypes and supertype.

Types and examples	Array name	File name
types, subtypes, supertypes eg. Person(p), Children(p)	types (eg. Person) type_ranges (eg. p)	st_expr_types.c st_typeranges.c

Table 3.2 : Arrays and files for storing types, subtypes, supertypes and objects

NOTE : The root type of the query is always stored in the first element of the array. This is done so that the Object Algebra expression generates the types of objects specified by the root type.

iii) predicate

This is a boolean expression consisting of function calls, comparison and boolean operators, or other SELECT statements. The predicate defined here is used as the predicate (F) of the Object Algebra expression. The set of objects will only be obtained if the predicate specified is true.

a) Queries with simple predicate (contains comparison and boolean operators) :

```
SELECT Name(p)
FOR EACH Person p, Children c
WHERE Name(c) = 'Tom' AND Name(c) IN Children c;
```

This query has the condition that the name of the child should be *Tom* and that he should be present in the subtype *Children* of the *Person* type. Therefore, the predicate (F) for the Object Algebra expression will be *<c>.Name = 'Tom' AND <c>.Name ∊ <c>.Children.*

Or another example could be

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Tasmania';
```

This query has the condition that the *Address* of the *Person* should be *Tasmania.*

NOTE : In Object Algebra expression "^" is AND and "∊" (ELEMENT of) is IN.

b) Nested queries, that is, predicate containing SELECT statement :
When another SELECT statement appears in the predicate clause of the previous SELECT statement, the translation is done in a slightly different way.

For example :

```
SELECT Name(p)
FOR EACH Person p
WHERE (Name(p)) IN
      (SELECT Name(c)
      FOR EACH Children c
      WHERE Age(c) >= 2);
```

This is a nested query and has the condition that the *Age* of the *Children* should be greater than or equal to 2.

The Object Algebra expression produced is as follows :

```
Person γ t [t ∈ <p>.Name ^ <c>.Name ∈
{Children γ t [t ∈ <c>.Name ^ <c>.Age >= '2']<>}]<>
```

The second SELECT statement, that is,

> **SELECT** Name(c)
> **FOR EACH** Children c
> **WHERE** Age(c) >= 2;

is translated into the Object Algebra expression (shown in {..}) and this becomes the predicate of the first SELECT statement.

NOTE : In the Object Algebra expression the target variables and the predicates are enclosed in the square brackets, that is, they will appear in [..].

All other nested queries are translated in a similar way, that is, each SELECT statement appearing in the predicate clause appears in {..} in the Object Algebra expression. The nested queries use the same variables but only increase the *level_nesting* to the required level.

Each of the functions, range variables, comparisons and boolean operators, and so on are also stored in separate files using different 4-dimensional arrays. *Table* 3.3 shows the array names and files for storing the predicate Identifiers.

Predicate types and example	Array name	File name
functions or types eg. Name(c), Age(c)	predfunc1 predfunc_range1	st_predfuncs.c st_predfuncranges.c
Comparison operators(=, <, >, <=, >=, <>) and IN and NOT_IN	Comp_operator1	comp_operators.c
Strings eg. 'Tasmaina', Alex	str1	st_strings.c
subtypes, types,supertypes eg. ....IN Children(c)	pred_type1 pred_type_range1	st_predtypes.c st_predtyperange.c
Operators(IN, NOT_IN, EXISTS NOT_EXISTS) in nested SELECTs	pred_op_type1	predfunc.c
functions, types for the nested SELECTs, that is, ...Name(p) IN (SELECT ....)	pred_in_func1 pred_in_func_range1	st_pred_infuncs.c st_pred_infuncranges.c

Table 3.3 : Arrays and files for storing predicates

Joining two or more subqueries with simple predicate or predicates containing SELECT statements could be as follows :

Two or more subqueries can be joined together using the UNION, INTERSECTION, and MINUS (DIFFERENCE) set operators. The translations of these queries are same as the above two queries (simple and nested) including only the set operator between the subqueries.

For example, the following query returns the names of all people who have a Child named *Tom* OR who live in *Tasmania*.

```
SELECT Name(p)
FOR EACH Person p, Children c
WHERE Name(c) = 'Tom' AND Name(c) IN Children c;
```

UNION

```
SELECT Name(p)
FOR EACH Person p
WHERE address(p) = 'Tasmania';
```

Translation

Once all the variables are stored in appropriate files as defined in the tables above, they are translated and printed out as they will appear in the Object Algebra expression. The printing of the SELECT query is done in the *printing1.c* file and printing.c file.

The ***printing1.c*** and ***printing.c*** files are the same except that printing1.c files prints the output, that is, the Object algebra expression onto an output file while the file printing.c simply prints it on the screen for the user to see it.

File : ***printing1.c***

```
/*********************************************************************************************************************/
include "variables.h"
#include <string.h>
#include <stdio.h>
int a;
FILE *ifp;

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

/* This procedure prints the introduction of each object algebra expression obtained
*/

introduction()
       {
       fprintf(ifp,"\n\n****************************************************");
       fprintf(ifp, "\n\nThe Object Algebra Expression of the above Query is :\n");
       }

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

/* This procedure prints the object algebra expression
*/
```

```
printing_OA_expression()
       {
       setl = 0;
       i = 0;
       a_level = 0;
       stat_level = 0;
       introduction();
       fprintf(ifp,"\n\n");
       print_OA_expr();
       check_for_next_select_statement();
       }
/
*********************************************************************/

print_OA_expr()
       {
       print_levels();
       print_rest_of_exprs();
       a_level = 0;
       }

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

print_levels()
       {
       fprintf(ifp, "%s  ", types[stat_level][a_level][0]);
       fprintf(ifp, "GEMMA (t) [t is an ELEMENT of ");
       print_specs();
       }

/********************************************************************/
/* This procedure checks the combinations of the result specification of the query
*/

print_specs()
       {
       if (functions[stat_level][a_level][0][0] != '\0')
            if (col_funcs[stat_level][a_level][0][0] != '\0')
                 if (constl[stat_level][a_level][0][0] != '\0')
                      if (other_exprl[stat_level][a_level][0]!='\0')
                           {
                           print_expr_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_func_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_const_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_other_expr_spec();
                           }
                      else
                           {
                           print_expr_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_func_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_const_spec();
                           }
                 else
                      if (other_exprl[stat_level][a_level][0]!='\0')
                           {
                           print_expr_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_func_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
                           print_other_expr_spec();
                           }
                      else
                           {
                           print_expr_spec();
                           fprintf(ifp," AND t is an ELEMENT of ");
```

```
                              print_func_spec();
                              }
          else
               if (constl[stat_level][a_level][0][0] != '\0')
                    if (other_exprl[stat_level][a_level][0]!='\0')
                              {
                              print_expr_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_const_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_other_expr_spec();
                              }
                    else
                              {
                              print_expr_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_const_spec();
                              }
               else
                    if (other_exprl[stat_level][a_level][0]!= '\0')
                              {
                              print_expr_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_other_expr_spec();
                              }
                    else
                              print_expr_spec();
     else
          if (col_funcs[stat_level][a_level][0][0] != '\0')
               if (constl[stat_level][a_level][0][0] != '\0')
                    if (other_exprl[stat_level][a_level][0]!='\0')
                              {
                              print_func_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_const_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_other_expr_spec();
                              }
                    else
                              {
                              print_func_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_const_spec();
                              }
               else
                    if (other_exprl[stat_level][a_level][0]!='\0')
                              {
                              print_func_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_other_expr_spec();
                              }
                    else
                              print_func_spec();
          else
               if (constl[stat_level][a_level][0][0] != '\0')
                    if (other_exprl[stat_level][a_level][0]!='\0')
                              {
                              print_const_spec();
                              fprintf(ifp," AND t is an ELEMENT of ");
                              print_other_expr_spec();
                              }
                    else
                              print_const_spec();
               else
                    if (other_exprl[stat_level][a_level][0]!= '\0')
                              print_other_expr_spec();

     check_predicate();
     check_for_next_level();
     }
```

```
/*********************************************************************/
/* This procedure translates other expressions of the result specification of the
query
*/

print_other_expr_spec()
      {
      ocl = 0;
      opl = 0;
      ofrl = 0;
      ofl = 0;
      oresl = 0;
      orangl = 0;
      orfl = 0;

      if (other_exprl[stat_level][a_level][0] == 1)
            fprintf(ifp,"(");
      for (oel = 1; oel < 100; oel++)
            {
            if (other_exprl[stat_level][a_level][oel] != '\0')
                  {
                  print_other_expr();
                  if (other_exprl[stat_level][a_level][oel] == 1)
                        {
                        ocl++;
                        opl++;
                        ofrl++;
                        ofl++;
                        oresl++;
                        orangl++;
                        orfl++;
                        fprintf(ifp,",(");
                        print_other_expr();
                        }
                  }
            }
      }

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

print_other_expr()
      {
      if (other_exprl[stat_level][a_level][oel] == 3)
            fprintf(ifp,"%s", other_consts[stat_level][a_level][ocl]);
      if (other_exprl[stat_level][a_level][oel] == 5)
            fprintf(ifp," %s ", other_ops[stat_level][a_level][opl]);
      if (other_exprl[stat_level][a_level][oel] == 2)
            {
            fprintf(ifp,"<%s>", other_func_ranges[stat_level][a_level][ofrl]);
            fprintf(ifp,".%s", other_funcs[stat_level][a_level][ofl]);
            }
      if (other_exprl[stat_level][a_level][oel] == 4)
            {
            fprintf(ifp,"%s", other_reswords[stat_level][a_level][oresl]);
            fprintf(ifp,"(<%s>", other_res_ranges[stat_level][a_level][orangl]);
            fprintf(ifp,".%s)", other_resfuncs[stat_level][a_level][orfl]);
            }
      if (other_exprl[stat_level][a_level][oel] == 6)
            fprintf(ifp,")");
      }

/*********************************************************************/
/* This procedure translates the constants of the result specification of the query
*/

print_const_spec()
      {
      if (constl[stat_level][a_level][1][0] == '\0')
            fprintf(ifp,"%s", constl[stat_level][a_level][0]);
```

```
        else
                {
                fprintf(ifp,"%s", constl[stat_level][a_level][0]);
                for (conl = 1; conl < 10; conl++)
                        if (constl[stat_level][a_level][conl][0] != '\0')
                                fprintf(ifp,",%s",constl[stat_level][a_level][conl]);
                }
        }

/*******************************************************************/
/* This procedure translates the functions, types, subtypes, supertypes of the
result specification of the query
*/

print_expr_spec()
        {
        if (functions[stat_level][a_level][1][0] == '\0')
                fprintf(ifp,"<%s>.%s", ranges[stat_level][a_level][0],
                functions[stat_level][a_level][0]);
        else
                {
                fprintf(ifp,"<%s>.(%s",
                ranges[stat_level][a_level][0],functions[stat_level][a_level][0]);
                for (f1 = 1; f1 < 10; f1++)
                        if (functions[stat_level][a_level][f1][0] != '\0')
                                fprintf(ifp,",%s", functions[stat_level][a_level][f1]);
                                fprintf(ifp,")");
                }
        }
/*******************************************************************/
/* This procedure translates the reserved functions of the result specification of
the query
*/

print_func_spec()
        {
        fprintf(ifp,"%s(<", res_funcs[stat_level][a_level][0]);
        fprintf(ifp,"%s>.", col_ranges[stat_level][a_level][0]);
        fprintf(ifp,"%s)", col_funcs[stat_level][a_level][0]);
        if (col_funcs[stat_level][a_level][1][0] != '\0')
                {
                crl = 0;
                rfl = 0;
                for (cl = 1; cl < 10; cl++)
                        {
                        crl++;
                        rfl++;
                        if (col_funcs[stat_level][a_level][cl][0] != '\0')
                                {
                                fprintf(ifp," AND t is an ELEMENT of ");
                                fprintf(ifp,"%s(<",res_funcs[stat_level][a_level][rfl]);
                                fprintf(ifp,"%s>.",
                                col_ranges[stat_level][a_level][crl]);
                                fprintf(ifp,"%s)", col_funcs[stat_level][a_level][cl]);
                                }
                        }
                }
        }

/***********************************************************************/
/* This procedure checks if there are any predicates or not
*/
check_predicate()
        {
        if (pred_spec != 0)
                print_pred_spec();
        }

/***********************************************************************/
/* This procedure translates the predicate of the Query
```

```
*/
print_pred_spec()
      {
      char in_op[1][128];
      char not_in_op[1][128];
      copl = 0;
      s1 = 0;
      pael = 0;
      pt = 0;
      pte = 0;
      strcpy(in_op[0], "IN");
      strcpy(not_in_op[0], "NOT_IN");
      for (pal = 0; pal < 5; pal++)
            if (predfuncl[stat_level][a_level][pal][0] != '\0')
                  {
                  fprintf(ifp," AND <%s>.",
                        predfunc_rangel[stat_level][a_level][pael]);
                  fprintf(ifp,"%s",  predfuncl[stat_level][a_level][pal]);
                  if(strcmp(in_op[0],
                        Comp_operatorl[stat_level][a_level][copl])==0)
                        {
                        fprintf(ifp," IS AN ELEMENT OF ");
                        fprintf(ifp,"<%s",
                        pred_type_rangel[stat_level][a_level][pte]);
                        fprintf(ifp,">.%s",pred_typel[stat_level][a_level][pt]);
                        pt++;
                        pte++;
                        }
                  else  if(strcmp(not_in_op[0],
                              Comp_operatorl[stat_level][a_level][copl])==0)
                        {
                        fprintf(ifp," IS NOT AN ELEMENT OF ");
                        fprintf(ifp,"<%s",
                        pred_type_rangel[stat_level][a_level][pte]);
                        fprintf(ifp,">.%s",pred_typel[stat_level][a_level][pt]);
                        pt++;
                        pte++;
                        }
                  else
                        {
                        fprintf(ifp,"%s",
                        Comp_operatorl[stat_level][a_level][copl]);
                        fprintf(ifp," %s ", strl[stat_level][a_level][s1]);
                        copl++;
                        s1++;
                        }
                  }
            pael++;
      }

/*************************************************************************/
/* This procedure translates other types, subtypes and supertypes.
*/

print_tables()
      {
      a_level = a;
      if (types[stat_level][a_level][1] == '\0')
            fprintf(ifp,"<>");
      else
            {
            fprintf(ifp,"<");
            fprintf(ifp,"%s", types[stat_level][a_level][1]);
            for (t1 = 2; t1 < 10; t1++)
                  if (types[stat_level][a_level][t1][0] != '\0')
                        fprintf(ifp,",%s", types[stat_level][a_level][t1]);
                  fprintf(ifp,">");
            }
      }
```

```
/*************************************************************************/
/* This procedure translates the next level of the SELECT statement in the nested
queries
*/

check_for_next_level()
        {
        char p_in_op[1][128];
        char p_not_in_op[1][128];
        i++;
        strcpy(p_in_op[0], "IN");
        strcpy(p_not_in_op[0], "NOT_IN");
        if(level_nesting[stat_level][i] ==1)
                {
                fprintf(ifp," AND ");
                fprintf(ifp,"<%s>.", pred_in_func_range1[stat_level][a_level][0]);
                fprintf(ifp,"%s", pred_in_func1[stat_level][a_level][0]);
                if(strcmp(p_in_op[0],pred_op_type1[stat_level][a_level][pf1])==0)
                        {
                        fprintf(ifp, " IS AN ELEMENT OF ");
                        pf1++;
                        }
                else
                        if (strcmp(p_not_in_op[0],pred_op_type1[stat_level][a_level]
                        [pf1])==0)
                                {
                                fprintf(ifp, " IS NOT AN ELEMENT OF ");
                                pf1++;
                                }
                a_level++;
                fprintf(ifp," \n{");
                print_levels();
                }
        }

/*************************************************************************/
/* This procedure translates the remaining expressions after printing the predicates
*/

print_rest_of_exprs()
        {
        for (a = a_level; a > 0; a--)
                {
                fprintf(ifp,"]");
                print_tables();
                fprintf(ifp,"}");
                }
        if (a == 0)
                {
                fprintf(ifp,"]");
                print_tables();
                }
        }

/*************************************************************************/
/* This procedure checks for the existence of other subqueries
*/

check_for_next_select_statement()
        {
        stat_level++;
        if (num_stats[stat_level] == 1)
                {
                fprintf(ifp," \n\n%s\n\n", set_op1[set1]);
                print_OA_expr();
                set1++;
                check_for_next_select_statement();
                }
        }
```

```
/**************************************************************************************************************************/
```

Dissection of *printing1.c* file

Procedure ***introduction()*** just prints the introduction.

Procedure ***printing_OA_expression()*** calls procedure ***print_OA_expr()***. It also checks if there are any more subqueries available.

Procedure ***print_OA_expr()*** checks if there are any SELECT statements to be translated. If there are then this procedure calls the procedures ***print_levels()*** and ***print_rest_of_exprs()***.

Procedure ***print_levels()*** translates the root type as the first argument of the GENERATE operator (represented by GEMMA). It then calls the procedure ***print_spec()***.

Procedure ***print_spec()*** checks the different combinations of the four types of result specifications : constants, functions, reserved functions, and other expressions containing operators. If there is some value in the arrays for storing the functions, constants, reserved functions, and other expressions then it means that the result specifications have that type of expression so it has to be translated. This procedure will then call other appropriate procedures to do the translations. The four procedures for the translations of the result specification (of Object SQL query) into the target variable, t (for the Object Algebra expressions) are : ***print_const_spec(), print_expr_spec(), print_func_spec(),*** and ***print_other_expr_spec()***. In this procedure the procedures ***check_predicate()*** and ***check_for_next_level()*** are also called.

Procedure ***print_const_spec()*** does the translation of the expressions with constants. All the constants are printed out one after the other.

Procedure ***print_expr_spec()*** does the translation of the simple expressions with list, tuple, set, or bag of functions, types, subtypes, supertypes, or simply functions, types, subtypes, supertypes separated by commas.

Procedure ***print_func_spec()*** does the translation of the expressions with reserved functions.

Procedure *print_other_expr_spec()* does the translation of the expressions with manipulation operators (for example, [5 * AVG(Salary(p)) + OverTimePay(P)] . To make the translations easier an array

other_expr1 [stat_level][a)level][oe1]

is used to assign a number to each of the open and closed square brackets, functions, constants, and reserved functions encountered. That is, if "[" is encountered then the number "1" is stored in the *other_expr1* array; if a function or type such as Name(p) is encountered then the number "2" is stored in the *other_expr1* array, and so on. *Table 3.4* shows what number is assigned to which of the symbols or expressions in the other expressions.

Identifier/Token	Number
[	1
function (eg. Salary(p))	2
Constant (eg. 4)	3
Reserved function (eg, AVG)	4
Operator (eg, *)	5
]	6

Table 3.4

During the translation this array is checked first to see how the expression should be printed out. It is important to note that these expressions are printed out as they appear in the result specification of the Object SQL query. Although the precedence of the operators has been implemented in the parser, it does not matter in the translation rules.

Procedure *check_predicates()* checks if the query has any predicates or not. If it has then the procedure *print_pred_spec()* is called.

Procedure *print_pred_spec()* translates the predicate part of the Object SQL query into the Object Algebra expression.

Procedure *print_tables()* translates all the types, subtypes, and supertypes of the Object SQL query.

Procedure *check_for_next_level()* checks for the presence of the nested queries.

Procedure *print_rest_of_exprs()* translates all the expressions after translating the predicate of the Object SQL query.

Procedure *check_for_next_select_statement()* checks if there are any subqueries.

3.4.2. Use of Translator

The following steps show how to obtain the Object Algebra expression using the parser (translator) written :

1) Firstly, compile the program using the make command.
2) Then run the program using object_sql.
3) At the 1>> prompt use the CREATE statement to create the functions, types, subtypes and supertypes. If this step is not done then the Object Algebra expression will not be produced. The message "Result Specification not in the database" will appear on the on the screen.
4) Then type the required Object SQL SELECT statement. The parser goes through each word and recognises each one according to the grammar rules specified in the YACC and LEX files.
5) After the whole query is parsed and there is no error the Object Algebra expression is printed out onto the output_file.
6) Open the output_file using the **vi** editor. Each of the SELECT statements is separated by a line of stars (*'s).

Notes :

1) Each time the program is run, a number of queries can be performed. However, each time a new program is run, that is, each time "object_sql" is typed, a new file is created. In other words the output_file is overwritten so if it is required to store the previous Object Algebra expressions the output_file should be copied to another file before running the program again.
2) During the actual printing of the Object Algebra expression in the output_file the symbol γ is replaced by the word "GEMMA" and the symbol $\in$ is replaced by the words "is an ELEMENT of".

Chapter 4. Results

The following examples cover most of the translations performed or handled by the translator. Appendix B gives the output result of some of these queries.

i) Some of the simple queries translated are :

Example 1 : Select the names of all people who live in Los Angeles.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Los Angeles';
```

The Object Algebra expression of the above query is :

Person γ^t[t $\in$ <p>.Name ^ <p>.Address = 'Los Angeles']<>

Example 2 : Select the names and age of all people. This is a tuple of 2 functions.

Object SQL SELECT QUERY

```
SELECT <|Name(p), Age(p)|>
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person γ^t[t $\in$ <p>.(Name, Age)]<>

Example 3 : Select the set of names, age, dateofbirth, salary of all people. This is a list of four functions.

Object SQL SELECT QUERY

```
SELECT {|Name(p), Age(p), DateOfBirth(p), Salary(p)|}
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.(Name, Age, DateOfBirth, Salary)]<>

Example 4 : Select the names of all people who have a child called Alex.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p, Children c
WHERE Name(c) = 'Alex' AND  Name(c) IN Children(p);
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.Name ∧ <c>.Name = 'Alex' ∧ <c>.Name ∈ <p>.Children]<Children>

Example 5 : Select the names and AVG(Salary(p)) of all people.

Object SQL SELECT QUERY

```
SELECT Name(p), AVG(Salary(p))
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.Name ∧ t ∈AVG(p.Salary)]<>

Example 6 : Select constant 5 of all people.

Object SQL SELECT QUERY

```
SELECT 5
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person $\gamma^t[t \in 5]<>$

Example 7 : Select constant 5 * Salary of all people.

Object SQL SELECT QUERY

```
SELECT [5 * Salary(p)]
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person $\gamma^t[t \in (5 * <p>.Salary)]<>$

Example 8 : Select the set of names, age, dateofbirth, and Children and Average Salary of all people.

Object SQL SELECT QUERY

```
SELECT {lName(p), Age(p), DateOfBirth(p), Children(p)l}, AVG(Salary(p))
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person $\gamma^t[t \in <p>.(\text{Name, age, DateOfBirth, Children}) \wedge t \in \text{AVG}(p.\text{Salary})]<>$

Example 9 : Select constant 5 * Salary + Overtime pay of all people.

Object SQL SELECT QUERY

```
SELECT [5 * Salary(p) + OvertimePay(p)]
FOR EACH Person p;
```

The Object Algebra expression of the above query is :

Person γ^t[t $\in$ (5 * <p>.Salary + <p>.OvertimePay)]<>

Example 10 : Select the names of all people who live in the State of Tasmania and in the City of Hobart and in the Street of Murray.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE State(p) = 'Tasmania' AND City(p) = 'Hobart' AND Street = 'Murray';
```

The Object Algebra expression of the above query is :

Person γ^t[t $\in$ <p>.Name AND <p>.State = 'Tasmania' AND <p>.City = 'Hobart' AND <p>.Street = 'Murray']<>

ii) Nested queries, that is, queries containing SELECT statements in its WHERE predicate.

Example 1 : Select the names of all people who have children greater than 10 years of age.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Age(c) > '10') ;
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.Name ^ <p>.Name ∈
{ Children γ^t[t ∈ <c>.Name ^ <c>.Age > '10']<> }]<>

Example 2 : Select the names of all people who have children whose name is not Tom.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) NOT_IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Name(c) = 'Tom');
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.Name ^ <p>.Name NOT ∈
{ Children γ^t[t ∈ <c>.Name ^ <c>.Name = 'Alex']<> }]<>

Example 3 : Select the names of all people and names of all the children whose age is less than two.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Age(c) NOT_IN
                ( SELECT Age(c)
                FOR EACH Children c
                WHERE Age(c) < '2')) ;
```

The Object Algebra expression of the above query is :

Person γ^t[t $\in$ <p>.Name ^ <p>.Name $\in$
{ Children γ^t[t $\in$ <c>.Name ^ <c>.Age NOT $\in$
{ Children γ^t[t $\in$ <c>.Age ^ <c>.Age < '2']<> }]<>}]<>

iii) Combining the result bag of 2 subqueries using UNION, INTERSECTION, MINUS, or DIVIDEBY.

Example 1 : Select the names of all people who live in either Los Angeles or in San Jose.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Los Angeles'

UNION

SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'San Jose';
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ <p>.Name ^ p.Address = 'Los Angeles']<>

UNION

Person γ^t[t ∈ <p>.Name ^ p.Address = 'San Jose']<>

Example 2 : Select the names of all people that receive either salaries of Teachers or salaries of Researchers.

Object SQL SELECT QUERY

```
SELECT Salary(r)
FOR EACH Researcher r

UNION

SELECT Salary(t)
FOR EACH Teacher t;
```

The Object Algebra expression of the above query is :

Researcher γ^t[t ∈ <r>.Salary]<>

UNION

Teacher γ^t[t ∈ <t>.Salary]<>

Example 3 : Select constant 5 * Salary of all people or select the names of all people who have children greater than 10 years of age.

Object SQL SELECT QUERY

```
SELECT [5 * Salary(p)]
FOR EACH Person p

UNION
```

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Age(c) > 10;
```

The Object Algebra expression of the above query is :

Person γ^t[t ∈ (5 * <p>.Salary)]<>

UNION

Person γt[t ∈ <p>.Name ^ <p>.Name ∈
{ Children γ^t[t ∈ <c>.Name ^ <c>.Age > 10]<> }]<>

Example 4 : Select the names of all people who have children whose name is not Tom or select the names of all people and names of all the children whose age is less than two.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE Name(p) NOT_IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Name(c) = 'Tom'

UNION

SELECT Name(p)
FOR EACH Person p
WHERE Name(p) IN
        ( SELECT Name(c)
        FOR EACH Children c
```

```
          WHERE Age(c) NOT_IN
                  ( SELECT Age(c)
                  FOR EACH Children c
                  WHERE Age(c) < '2';
```

The Object Algebra expression of the above query is :

Person γ^{t}[t ∈ <p>.Name ∧ <c>.Name IS NOT AN ∈
{ Children γ^{t}[t ∈ <c>.Name ∧ <c>.Name = 'Alex']<> }]<>

UNION (U)

Person γ^{t}[t ∈ <p>.Name ∧ <p>.Name ∈
{ Children γ^{t}[t ∈ <c>.Name ∧ <c>.Age > 10 AND <c>.Age NOT ∈
{ Children γ^{t}[t ∈ <c>.Age ∧ <c>.Age < '2']<> }]<>}]<>

> **Example 5** : Select the names of all people who live in the State of Tasmania and the City of Hobart.

Object SQL SELECT QUERY

```
SELECT Name(p)
FOR EACH Person p
WHERE State(p) = 'Tasmania'

INTERSECTION

SELECT Name(p)
FOR EACH Person p
WHERE City(p) = 'Hobart';
```

The Object Algebra expression of the above query is :

Person γ^{t}[t ∈ <p>.Name ∧ <p>.State = 'Tasmania']<>

INTERSECTION

Person γ^{t}[t $\in$ <p>.Name $\wedge$ <p>. City = 'Hobart']<>

Chapter 5. Conclusion

5.1. Achievements

The main aim of this project was to write a translator from Object SQL into the Object Algebra expression. The project was worked on from early February 1993 to mid January 1994. During this period the following goals were accomplished :

a) a working definition of the Object SQL statements, that is, a parser for the Object SQL was written.

b) Using Straube's object Algebra [Straube 90a], the rules for translations of Object SQL statements into Object Algebra expressions were written. (The selection of the Object Algebra to be used in this project was done by a Masters student as part of her thesis.)

c) Then using the translation rules and the working parser, a working translator was written. The translator uses the Object SQL query statement as input and produces an Object Algebra expression as the output.

The results of the translations from Object SQL into the Object Algebra expressions is presented in Chapter 4 of this thesis. The implementation methods are discussed in Chapter 3.

The translation of the logical operator "AND" was performed but not the translations for the logical operator "OR". This was so because the "UNION" operator can be used to combine two subqueries which will have the same effect as having an "OR" in the predicate of the Object SQL query statement.

When doing the nested queries, only the "IN" and "NOT IN" operators were implemented. To make the implementation easier the "NOT IN" is used as a single word "NOT_IN".

Clauses like GROUP BY, ORDER BY, and HAVING were not translated. Also, since the translation of other Object SQL statements such as UPDATE, CREATE, DELETE, INSERT are not as well defined for objects stores as for relational databases, translation of these was not performed.

5.2. Discussion/Conclusions

Although this project was a continuation of a 1992 honours project in which the Object SQL parser was written, the parser was not used. Instead a completely new parser was written.

Since the official (Hewlett-Packard) Object SQL grammar does not exist yet, there is no guarantee that the grammar rules that I have written will closely resemble those written by Hewlett-Packard.

After some initial problems in using the YACC and LEX to write the parser I found that YACC and LEX provided a convenient form of making the Object SQL compiler. One of the major problems encountered during the writing of the parser was the "conflicts (shift/reduce or reduce/reduce)" in some of the rules. Because there were so many rules at times it was hard to get rid of some of the shift/reduce conflicts. The rules that had the conflicts had to be studied very carefully to make sure that the right result was obtained.

Also the use of C language with YACC and LEX to write the translation rules was very convenient. During the course of this project I learnt a lot about C, YACC and LEX programming.

A data dictionary was created before doing the translations of the query.

Since an Object SQL standard does not exist yet, the existing Leroy Cain's SQL compiler provided a convenient starting point for the Object SQL compiler. With slight modifications some of Leroy Cain's [Cain 1989] files used in the SQL parser were used in the Object SQL parser.

The Object Algebra (by Straube) used in this project was a more convenient target language than other Object Algebras such as the one provided by Zdonik and Shaw [Shaw 1989]. Also fortunately the Object Algebra used had an operator GENERATE which could be used in the translation of the Object SQL into the Object Algebra expression.

Dave Straube's Object Algebra was used to fulfil the role of a complete internal representation of the Object SQL queries. This enabled the subset of Object SQL queries to be translated into the Object Algebraic expressions.

Some of the features that Straube's object algebra supports are :

1) *Object-Orientedness* (supports identities, encapsulation, inheritance hierarchy, and heterogenous sets).
2) *Formal Properties* (a formal semantics and a closed algebra).
3) *Performance* (provides optimisation strategies and supports strong typing).
4) *Database Issues* (has an equivalent object calculus).

Also, Straube's Object Algebra has a very detailed mathematical definition of its operations. The proofs of all these operations are also provided. This is very important so that the meaning of the Object Algebra is very clear and can be easily used in the query optimisation process.

5.3 Summary

This project involved OODBMS, new technology that is still in the early stages of the development process. In the next couple of years OODBMS is likely to become a very common and widely used database system. Object-Oriented databases and languages have some major advantages over relational databases. They tend to provide a more structured, consistent, and powerful base than their older counterparts.

Some of the important features that OODBMS allows are :

- complex type definitions.
- encapsulation of information in a package-like environment, removing the possibility of non-standard functions accessing private data.
- ability to overload methods, providing convenience for programmers who then do not have to remember extra function names.
- abstraction of implementation away from the end-users.

For these reasons OODBMS is appearing as one of the leading technologies for this decade.

Object SQL is used as a query language for the OODBMS. It is a subset of the ANSI SQL standard with object processing extensions. It is not tied to a specific language but is intended to combine the best from object-oriented programming languages with the benefits of declarative data definition and manipulation languages of SQL.

Nowadays, query algebra is gaining a lot of attention in the database area. Researchers are trying to define a query model and an Object-Oriented algebra that will help in query optimisation.

Some of the properties of the query languages are : object-orientedness, expressiveness, formal properties, performance, and database issues.

5.3. Future Work

One of the major criticisms of most database management systems is their lack of efficiency in handling the powerful operations they offer. This can be particularly seen on gaining data accesses through the use of queries, where it is essential to provide immediate responses to user requests. Query optimisation attempts to solve this problem. Hence query optimisation is seen as one of the important aspects of query languages.

The translator from Object SQL into the Object Algebra expression written in this project can be used in the optimisation process. Most of the translation rules have been written. The translation of clauses like GROUP BY, ORDER BY, and HAVING of the Object SQL query statement has not yet been implemented. The Object Algebra used in this project needs to be extended to include these translations.

In the implementation of the translator the data dictionary used for storing the functions and types is an array. This data dictionary could be divided into a data dictionary containing the functions and a data dictionary containing the types.

In the result specification it is assumed that the range variables for all the functions in the list, set, etc, are the same. That is, for example, at the moment the result specification should have the following form :

{| Name(p), Age(p), Birthdate(p) |}

If it is not in the above form the translator will not give an error message and the Object Algebra expression produced will not be correct. However, this translation rule could be modified further to check for the range variables of different types, subtypes and supertypes.

Furthermore, some more operators need to do added to Straube's Object Algebra to translate the DELETE, INSERT, UPDATE, and CREATE statements.

Appendix A: Object SQL Parser and Translator

A.1. Explanation of Symbols used in EBNF Grammar

Table A.1 summarises some of the symbols used in the EBNF grammar rules.

Symbol	Symbol Meaning
WORD	Terminal symbol (reserved word)
word	Non-terminal symbol, that is, a 'pointer' to another rule in the grammer
[ ]	Optional
( )	Grouping
|	Or
"s", 's'	If s is a symbol, then it is a non-terminal, otherwise, it is a symbol having the same character sequence as in the input.
( )*	Zero or more
( )+	One or more
{ }	C procedures for rule translations

Table A.1 : Symbols used in EBNF grammar

In the following section the Extended Backus Naur Form (EBNF) of the Object SQL language (as defined in the Object SQL parser) is presented.

A.2. Grammar for Object SQL parser

NOTE : Although not shown, each rule terminates with a semicolon.

```
object_sql_internal_statement ::= (stats)+
        | RW_STORE RW_PROGRAM prog_name '(' parm_list ')' (stats)+ RW_END
        RW_STORE

stats ::= states (';' states)*
```

```
states ::= [ses_var ':'] osql_statement

parm_list ::= parm (',' parm)*

parm ::= parm_name ['=' expr RW_ASSIGN]

parm_name ::= IDENTIFIER

initialization_of_arrays ::=
        {
        init_of_arrays();
        }

initializationOf_num_stats ::=
        {
        for (j=0; j <=10; j++)
                num_stats[j] = 0;
        }

initializationOf_l_nesting ::=
        {
        for (j=0; j <=10; j++)
                level_nesting[j][0] = 0;
        }

osql_statement ::= add | alter | audit | begin | check | close | comment | commit |
        connect | create | delete | disconnect | drop | dump | fetch | grant | import | info
        | insert | lock | open | remove | repair | restore | revoke | rollforward |
        rollback | savepoint | initializationOf_num_stats initializationOf_l_nesting
        {sel_stat = 0;} select | ses_var | set | shell | start | sync | update | export |
        SCRIPT

create ::= RW_CREATE RW_TYPE multi_valued_objects [sub_type function_defs]
        | RW_CREATE type view_name [vfd] RW_AS select_statement [with_clause]
        | RW_CREATE RW_SYNONYM syn_name RW_FOR path
        | RW_CREATE unique RW_CLUSTER RW_INDEX index_name RW_ON path
        | RW_CREATE RW_DATABASE dbname RW_TYPE dbtype RW_ON hname
        [with_clause]
        | RW_CREATE RW_DOMAIN domain_name_list RW_IS domain_types defaults
        domain_restricts
        | RW_CREATE RW_CONSTRAINT constr_name trigger_list RW_CHECK
        predicate
        | RW_CREATE path_element RW_FUNCTION '(' type_name_list ')' var_list
        | RW_CREATE object_name p_list
        | cs RW_FUNCTION path_element '(' t_list ')' [r_type]
        | RW_CREATE RW_CLUSTER RW_TABLE path tfd_list [with_clause] '('
        order_list ')' [with_clause]

p_list ::= ses_var (ses_vars)*

cs ::= RW_CREATE
        | RW_IMPLEMENT

t_list ::= type_name_list
        | fp_list

fp_list ::= var_decl (',' var_decl)*

r_type ::= RW_AS as_list

tfd_list ::= [primary_keys] [foreign_keys] field_defs

primary_keys ::= RW_PRIMARY '(' primary_keys ')'

primary_keys ::= field (',' field)*

foreign_keys ::= foreign_keys foreign_key

foreign_key ::= RW_FOREIGN '(' f_key IDENTIFIER path nulls
RW_DELETE RW_OF effect RW_UPDATE RW_OF path effect

f_key ::= IDENTIFIER

nulls ::= RW_NULL RW_ALLOWED
        | RW_NULL RW_NOT RW_ALLOWED

field_defs ::= '(' field_definitions ')'
```

```
field_definitions ::= single_definition (',' single_definition)*

single_definition ::= var_decl [declaration_option] [opt_null]

var_decl ::= type_name vname

type_name ::= IDENTIFIER
        {
        s_type[0][0] = '\0';
        strcpy(s_type[0], yytext);
        store_types();
        }
        [type_elm]
        | aggregate

type_elm ::= IDENTIFIER

aggregate ::= tuplet | bagt | sett | listt

tuplet ::= "<|" type_name_list "|>"

bagt ::= "[:" type_name_list ":]"

sett ::= "{|" type_name_list "|}"

listt ::= "[|" type_name_list "|]"

type_name_list ::= type_name (',' type_name)*

declaration_option ::= RW_FUNCTION
        | '(' expr ')'

opt_null ::= nulls

view_name ::= [user '.'] record

record ::= IDENTIFIER
```

```
syn_name ::= IDENTIFIER

order_list ::= path orderdir RW_ORDER (',' path orderdir)*

dbtype ::= IDENTIFIER
        | PARM

hname ::= IDENTIFIER

domain_name_list ::= domain_name (',' domain_name)*

domain_types ::= domain_type
        | '(' domain_type_list ')'

domain_type ::= type_name ('(' i_list ')')*

i_list ::= INTEGER (',' INTEGER)*

domain_type_list ::= domain_type (',' domain_type)*

defaults ::= RW_DEFAULT

domain_restricts ::= where_clause
        | select_statement

constr_name ::= IDENTIFIER

trigger_list ::= trigger (',' trigger)*

trigger ::= RW_AT event
        | RW_AFTER event RW_OF path RW_FROM path
        | RW_BEFORE event RW_OF path RW_FROM path

event ::= RW_COMMIT
        | RW_DELETE
        | RW_INSERT
        | RW_UPDATE

fexpr ::= select_expr
```

```
        | insert
        | update
        | expr
        | block

block ::= RW_BEGIN create RW_END

type ::= RW_VIEW
        | RW_FRAGMENT
        | RW_SNAPSHOT

sub_type ::= RW_SUBTYPE RW_OF multi_valued_objects

multi_valued_objects ::= m_object (',' m_object)*

m_object ::= IDENTIFIER
        {
        s_type[0][0] = '\0';
        strcpy(s_type[0], yytext);
        store_types();
        }

function_defs ::= RW_FUNCTION '(' funct_list ')'

funct_list ::= m_object [as_clause] (',' m_object [as_clause])*

as_clause ::= RW_AS as_list

as_list ::= RW_FORWARD | RW_FOREIGN | RW_DERIVED | RW_PROCEDURAL |
        RW_STORED | RW_EXTERNAL | RW_STANDARD | fexpr

select ::=
        {
        sel_stat = 1;
        }
        initialization_of_arrays
        select_expr [order_clause]
        {
        introduction();
        printing_OA_expression();
        pred_spec = 0;
        func_or_type = 0;
        printf("\n")
        }

order_clause ::= RW_ORDER RW_BY sort_specification_list

sort_specification_list ::= sort_specification (',' sort_specification)*

sort_specification ::= IDENTIFIER '(' IDENTIFIER ')' orderdir

orderdir ::= RW_ASC
        | RW_DESC

select_expr::= inc_num_stats inc_level_nesting select_statement (m_select_expr)*

m_select_expr ::= inc_num_stats inc_level_nesting RW_UNION st_set_OP_type
        [any] select_statement
        | RW_MINUS select_statement
        | RW_DIVIDEBY select_statement
        | inc_num_stats inc_level_nesting RW_INTERSECT st_set_OP_type
        select_statement

st_set_OP_type ::=
        {
        set_op[0][0] = '\0';
        strcpy(set_op[0], yytext);
        store_set_ops();
        }

any ::= RW_ANY
        | RW_ALL

select_statement ::=
        select_clause RW_FOR RW_EACH table_list [select_options]

select_options ::= where_clause
        | group_clause
```

```
group_clause ::= RW_GROUP RW_BY IDENTIFIER '(' IDENTIFIER ')' having

having ::= RW_HAVING hav_clause

hav_clause ::= IDENTIFIER '(' IDENTIFIER ')' c_ops ints

ints ::= INTEGER
        | REAL

c_ops ::= '=' | "<>" | '>' | '<' | ">=" | "<="

select_clause::= RW_SELECT sellist
        | RW_SELECT RW_UNIQUE sellist
        | RW_SELECT unique '*'

sellist ::= sel_expr_type (',' sel_expr_type)*

sel_expr_type ::= sel_expression
        | function_list
        | ses_var
        | const
        | other_expressions

sel_expression ::= expr
        | "<|" expr (',' expr)* "|>"
        | "[|" (',' expr)* "|]"
        | "[:" expr (',' expr)* ":]"
        | "{|" expr (',' expr)* "|}"

expr ::= expr1 '(' elm_type ')'

elm_type ::= elm1
        | ses_var

other_expressions ::= '[' {store_stat_lbracket(); } expr2 (type_op)+ ']'
        {store_stat_rbracket(); }

type_op ::= '+'
```

```
        {
        other_op[0][0] = '\0';
        strcpy(other_op[0], yytext);
        store_stat_other_ops();
        }
        expr2
        | '-'
        {
        other_op[0][0] = '\0';
        strcpy(other_op[0], yytext);
        store_stat_other_ops();
        }
        expr2
        | '*'
        {
        other_op[0][0] = '\0';
        strcpy(other_op[0], yytext);
        store_stat_other_ops();
        }
        expr2
        | '/'
        {
        other_op[0][0] = '\0';
        strcpy(other_op[0], yytext);
        store_stat_other_ops();
        }
        expr2

expr2 ::= exp2 '(' elm2 ')'
        | INTEGER
        {
        other_const[0][0] = '\0';
        strcpy(other_const[0], yytext);
        store_stat_other_consts();
        }
        | REAL
        | RW_AVG
        {
        other_resword[0][0] = '\0';
        strcpy(other_resword[0], yytext);
        store_stat_other_resword();
        }
        '(' col_spec ')'
        | RW_SUM
        {
        other_resword[0][0] = '\0';
        strcpy(other_resword[0], yytext);
        store_stat_other_resword();
        }
        '(' col_spec ')'
        | RW_MAX
        {
        other_resword[0][0] = '\0';
        strcpy(other_resword[0], yytext);
        store_stat_other_resword();
        }
        '(' col_spec ')'
        | RW_MIN
        {
        other_resword[0][0] = '\0';
        strcpy(other_resword[0], yytext);
        store_stat_other_resword();
        }
        '(' col_spec ')'

col_spec ::= colm_name '(' colm_elm ')'

exp2 ::= IDENTIFIER
        {
        other_func[0][0] = '\0';
        strcpy(other_func[0], yytext);
        store_stat_other_funcs();
        }

elm2 ::= IDENTIFIER
        {
        other_func_range[0][0] = '\0';
        strcpy(other_func_range[0], yytext);
        store_stat_other_func_ranges();
        }

colm_name ::= IDENTIFIER
        {
        other_resfunc[0][0] = '\0';
        strcpy(other_resfunc[0], yytext);
        store_stat_other_res_func();
        }

colm_elm ::= IDENTIFIER
        {
        other_res_range[0][0] = '\0';
        strcpy(other_res_range[0], yytext);
        store_stat_other_res_ranges();
        }

function_list ::= set_function_specification

set_function_specification ::= distinct_set_function

distinct_set_function ::= RW_AVG st_res_func '(' column_specification ')'
        | RW_MAX st_res_func '(' column_specification ')'
        | RW_MIN st_res_func '(' column_specification ')'
        | RW_SUM st_res_func '(' column_specification ')'
        | RW_COUNT st_res_func '(' column_specification ')'

st_res_func ::=
        {
        res_func[0][0] = '\0';
        strcpy(res_func[0], yytext);
        store_res_funcs();
        }

column_specification ::= column_name '(' column_elm ')'

column_name ::= IDENTIFIER
        {
        if (sel_stat == 1)
```

```
                {
                col_func[0][0] = '\0';
                strcpy(col_func[0], yytext);
                m1 = 0;
                check_col_func();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

column_elm ::= IDENTIFIER
        {
        col_range[0][0] = '\0';
        strcpy(col_range[0], yytext);
        store_colranges();
        }

expr1 ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                function[0][0] = '\0';
                strcpy(function[0], yytext);
                m1 = 0;
                check_func_or_types();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

elm1 ::= IDENTIFIER
        {
        range[0][0] = '\0';
        strcpy(range[0], yytext);
        store_function_ranges();
        }

unique ::= RW_ALL
        | RW_DISTINCT
        I RW_UNIQUE

table_list ::= table1_name table_elm1 (',' table1_name table_elm1)*

table_elm1 ::= IDENTIFIER
        {
        type_range[0][0] != '\0';
        strcpy(type_range[0], yytext);
        store_type_ranges();
        }

table1_name ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                type[0][0] != '\0';
                strcpy(type[0], yytext);
                m1 = 0;
                check_types();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

where_clause ::= RW_WHERE {pred_spec = 0} predicate {pred_spec = 1}
        I RW_WHERE '(' p_in_atts ')' p_in_op_type inc_level_nesting '('
        select_statement ')'

p_in_atts ::= p_atts '(' p_attelm ')' (',' p_atts '(' p_attelm ')')*

p_atts ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                pred_in_att[0][0] = '\0';
                strcpy(pred_in_att[0], yytext);
                m1 = 0;
                check_pred_in_att();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

p_attelm ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                pred_in_att_elm[0][0] = '\0';
                strcpy(pred_in_att_elm[0], yytext);
                st_stat_pred_in_att_elm();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

p_in_op_type ::= pred_in_op_type
        I pred_exist_op_type

pred_in_op_type ::= RW_IN st_pred_in_op_type
        I RW_NOT_IN st_pred_in_op_type

pred_exist_op_type ::= RW_EXISTS
        I RW_NOT_EXISTS

st_pred_in_op_type ::=
        {
        pred_op_type[0][0] = '\0';
        strcpy(pred_op_type[0], yytext) ;
        store_pred_func();
        }

inc_level_nesting ::=
        {
        int temp_nesting;
        temp_nesting = j + 1;
        if ((num_stats[j] == 1) && (num_stats[temp_nesting]==0))
                {
                stat_level = j;
                i = 0;
                while((i < 10) && (level_nesting[stat_level][i] != 0))
                        i++;
                if (i == 10)
                        printf("ERROR in level of nesting");
                else
                        level_nesting[stat_level][i] = 1;
                }
        }

inc_num_stats ::=
        {
        j = 0;
        while((j < 10) && (num_stats[j] != 0))
                j++;
        if (j == 10)
                printf("ERROR in number of select statement");
        else
                num_stats[j] = 1;
        }

predicate ::= op_predicate (RW_AND op_predicate)*

op_predicate ::= predattrib1 '(' predelm1 ')' comp_op predicate_type

predicate_type ::= string1
        | pred_table '(' pred_tble_elm ')'

pred_table ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                pred_type[0][0] = '\0';
                strcpy(pred_type[0], yytext);
                m1 = 0;
                check_pred_types();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

pred_tble_elm ::= IDENTIFIER
        {
        pred_type_elm[0][0] = '\0';
        strcpy(pred_type_elm[0], yytext);
        store_pred_type_elms();
        }

predattrib1 ::= IDENTIFIER
        {
        if (sel_stat == 1)
                {
                predatt[0][0] = '\0';
                strcpy(predatt[0], yytext);
                m1 = 0;
                check_pred_funcs();
                }
        else
                printf("....NOT SELECT STATEMENT....");
        }

predelm1 ::= IDENTIFIER
        {
        predatt_element[0][0] = '\0';
        strcpy(predatt_element[0], yytext);
        store_predattelms();
        }

comp_op ::= '=' st_ops | "<>" st_ops | '<' st_ops | "<=" st_ops | ">=" st_ops | '>'
        st_ops | RW_IN st_ops

st_ops ::=
        {
        Comp_operator[0][0] = '\0';
        strcpy(Comp_operator[0], yytext);
        store_operator();
        }
```

```
string1 ::= STRING
        {
        str[0][0] = '\0';
        strcpy(str[0], yytext);
        store_strings();
        }

savepoint ::= ses_var ":=" RW_SAVEPOINT

connect ::= RW_CONNECT RW_TO dbname

disconnect ::= RW_DISCONNECT RW_FROM dbname
        | RW_DISCONNECT dbname

dbname ::= IDENTIFIER

begin ::= RW_BEGIN [work]

work ::= RW_WORK

commit ::= RW_COMMIT [work]

rollback ::= RW_ROLLBACK [work]
        | RW_ROLLBACK sync

sync ::= RW_SYNC

check ::= RW_CHECK

dump ::= RW_DUMP RW_DATABASE dbname [to_clause]
        | RW_DUMP RW_TABLE dbname [to_clause]

to_clause ::= RW_TO filename

filename ::= IDENTIFIER

repair ::= RW_REPAIR

restore ::= RW_RESTORE RW_DATABASE dbname [rfrom_clause]
```

```
        I RW_RESTORE RW_TABLE dbname [rfrom_clause]

rfrom_clause ::= RW_FROM filename

fetch ::= RW_FETCH cname [next]

next ::= RW_NEXT INTEGER

cname ::= IDENTIFIER

close ::= RW_CLOSE dbname

open ::= RW_OPEN dbname [cursoropt]

cursoropt ::= password
        I RW_FOR select_statement

password ::= IDENTIFIER

ses_var ::= SESVAR

shell ::= RW_SHELL '(' STRING ')'

info ::= RW_INFO

remove ::= RW_REMOVE RW_TYPE tname RW_FROM rem_list

tname ::= IDENTIFIER

rem_list ::= reme1 (',' reme1)*

reme1 ::= ses_var
        I vname

vname ::= IDENTIFIER

export ::= RW_EXPORT object_type object_name [into_clause]
```

```
object_type ::= RW_DATABASE | RW_DOMAIN | RW_PROGRAM | RW_SYNONYM |
        RW_TYPE | RW_TABLE | RW_VIEW | RW_FUNCTION

object_name ::= IDENTIFIER
        {
        s_type[0][0] = '\0';
        strcpy(s_type[0], yytext);
        store_types();
        }

into_clause ::= RW_INTO filename

set ::= RW_SET option_list

option_list ::= option (',' option)*

option ::= IDENTIFIER
        | IDENTIFIER IDENTIFIER
        | IDENTIFIER '=' const

const ::= INTEGER st_consts
        | REAL | STRING | RW_DEFAULT | RW_NULL | PARM

st_consts ::=
        {
        const[0][0] = '\0';
        strcpy(const[0], yytext);
        store_consts();
        }

import ::= RW_IMPORT object_type object_name rfrom_clause

revoke ::= RW_REVOKE [privilege] [gr_on] RW_FROM users
        | RW_REVOKE RW_UPDATE RW_ON RW_FUNCTION [privilege] RW_FROM
        RW_PUBLIC

privilege ::= (privilig)+

privilig ::= priv_name ('(' field_list ')')*
```

```
priv_name ::= IDENTIFIER | RW_ALTER | RW_DELETE | RW_DUMP | RW_RESTORE |
        RW_DROP | RW_INSERT | RW_LOCK | RW_SELECT | RW_UPDATE | RW_CALL |
        RW_ALL

field_list ::= fld (',' fld)*

fld ::= IDENTIFIER '(' field ')'
        | field

field ::= IDENTIFIER

gr_on ::= RW_ON object_type objects

objects ::= object_name (',' object_name)*

users ::= user (',' user)*

user ::= IDENTIFIER
        | PARM

lock ::= RW_LOCK RW_TABLE path RW_IN mode RW_MODE

path ::= path_e_list ('(' element_name ')')*

path_e_list ::= path_element ('.' path_element)*

element_name ::= norder_list

norder_list ::= expr orderdir RW_ORDER (',' expr orderdir RW_ORDER)*

path_element ::= IDENTIFIER
        {
        s_type[0][0] = '\0';
        strcpy(s_type[0], yytext);
        store_types();
        }

mode ::= IDENTIFIER | RW_SHARE | RW_EXCLUSIVE | RW_ALL
```

```
audit ::= RW_AUDIT [into_clause] path_list [from_clause] [where_clause1]

path_list ::= path (',' path)*

from_clause ::= RW_FROM path_list

where_clause1 ::= RW_WHERE predicate

drop ::= RW_DROP RW_INDEX index_name
        | RW_DROP object_type path

index_name ::= path

comment ::= RW_COMMENT RW_ON path
        | RW_COMMENT RW_ON path RW_IS comment_str

comment_str ::= STRING (',' STRING)*

alter ::= RW_ALTER path alter_list [with_clause]
        | RW_ALTER RW_TABLE path [with_clause]

with_clause ::= RW_WITH option_list

alter_list ::= alteration (',' alteration)*

alteration ::= RW_ADD '(' a_tfd_fields ')'
        | RW_DROP [vfd]
        | RW_MODIFY '(' m_tfd_fields ')'

a_tfd_fields ::= tfd [before] (',' tfd [before])*

tfd ::= field [o_domain_name]
        | field '(' field_list ')' [o_domain_name]

o_domain_name ::= domain_name
        | function

domain_name ::= IDENTIFIER
```

before ::= RW_BEFORE field

vfd ::= '(' vfd_list ')'

vfd_list ::= field (',' field)*

m_tfd_fields ::= field tfd (',' field tfd)*

rollforward ::= RW_ROLLFORWARD path RW_FROM path [roll_op]

roll_op ::= RW_TO STRING [time]

time ::= STRING

grant ::= RW_GRANT [privilege] [gr_on] RW_TO users [at_option] [between_option]
        [on_option] [where_option] [wgo]
        | RW_GRANT RW_CALL RW_ON RW_FUNCTION path_element period_elm
        RW_TO users [at_option] [between_option]

at_option ::= RW_AT terminal_list

terminal_list ::= tty (',' tty)*

tty ::= IDENTIFIER

between_option ::= RW_BETWEEN time1 RW_AND time2

time1 ::= IDENTIFIER

time2 ::= IDENTIFIER

on_option ::= RW_ON day1 RW_AND day2

day1 ::= IDENTIFIER

day2 ::= IDENTIFIER

where_option ::= RW_WHERE predicate

```
wgo ::= RW_WITH RW_GRANT RW_OPTION

insert ::= RW_INSERT RW_INTO rec_alias '(' field_list ')' icond
        | RW_INSERT RW_INTO rec_alias icond

rec_alias ::= ses_var
        | path
        | path alias

alias ::= IDENTIFIER

icond ::= RW_FROM filename
        | select_statement
        | RW_VALUES '(' expr_list ')'

expr_list ::= expr (',' expr)*

delete ::= RW_DELETE RW_FROM rec_alias [where_clause]
        | RW_DELETE RW_TYPE path doto
        | RW_DELETE RW_FUNCTION path_element [period_elm] [doto]
        | RW_DELETE ses_var

period_elm ::= '.' IDENTIFIER

doto ::= effect

effect ::= RW_CASCADE
        | RW_RESTRICTED | RW_NULLIFIES | RW_ALL

add ::= RW_ADD RW_TYPE tname RW_FUNCTION '(' field_list ')' RW_TO var_list

var_list ::= sing_var (',' sing_var)*

sing_var ::= ses_var '(' e_list ')'

e_list ::= STRING
        | INTEGER
        | '[' ses_var (',' ses_var)* ']'
```

```
start ::= prog_name '(' expr_list ')'

prog_name ::= IDENTIFIER

update ::= RW_UPDATE rec_alias [set_show] where_clause
        | RW_UPDATE path_element '(' ses_var ')' ":=" update_type

update_type ::= STRING | INTEGER | REAL

set_show ::= RW_SHOW set_elements
        | RW_SET set_elements

set_elements ::= set_element (',' set_element)*

set_element ::= fld '=' expr RW_ASSIGN
        | fld '=' select_statement RW_ASSIGN
        | '(' field_list ')' '=' '(' expr_list ')' RW_ASSIGN
        | '(' field_list ')' '=' select_statement RW_ASSIGN
```

Appendix B: Output file for the translator

This is the output file which shows some of the translations (of the Object SQL statements) done by the translator. This output file only contains the Object Algebra expressions. The Object SQL query statements are shown to make the reading of the Object Algebra expressions easier.

```
************************************************************
Object SQL Query :
SELECT Name(p)
FOR EACH Person p;


The Object Algebra Equivalent of the Query is :
Person  GEMMA (t) [t is an ELEMENT of <p>.Name]<>

************************************************************
Object SQL Query :
SELECT {|Name(p), Age(p), Children(c), Salary(p)|}
FOR EACH Person p;


The Object Algebra Equivalent of the Query is :
Person  GEMMA (t) [t is an ELEMENT of <p>.(Name,Age,Children,Salary)]<>

************************************************************
Object SQL Query :
SELECT 5
FOR EACH Person p;


The Object Algebra Equivalent of the Query is :
Person  GEMMA (t) [t is an ELEMENT of 5]<>

************************************************************
Object SQL Query :
SELECT [5 * Salary(p)]
FOR EACH Person p;
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of (5 * <p>.Salary)]<>

****

Object SQL Query :

```
SELECT Name(e), AVG(Salary(e))
FOR EACH Employee e;
```

The Object Algebra Equivalent of the Query is :

Employee GEMMA (t) [t is an ELEMENT of <p>.Name AND t is an ELEMENT of AVG(<e>.Salary)]<>

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Tasmania';
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Address= 'Tasmania']<>

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Tasmania' AND Salary(p) > '2000';
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Address= 'Tasmania' AND <p>.Salary> '2000']<>

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
```

```
WHERE (Name(c)) IN
        ( SELECT Name(c)
        FOR EACH Children c
        WHERE Age(c) > '10' );
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <c>.Name
{Children GEMMA (t) [t is an ELEMENT of <c>.Name AND <c>.Age > '10']<>}]<>

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
WHERE (Name(p)) IN
        (SELECT Name(c)
        FOR EACH Children c
        WHERE (Age(c)) IN
                (SELECT Name(c)
                FOR EACH Children c
                WHERE Age(c) > '10'));
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Name is an ELEMENT of
{Children GEMMA (t) [t is an ELEMENT of <c>.Name AND <c>.Age is an ELEMENT of
{Children GEMMA (t) [t is an ELEMENT of <c>.Name AND <c>.Age > '10']<>}]<>}]<>

****

Object SQL Query :

```
SELECT Name(r)
FOR EACH Researcher r
WHERE Salary(r) > '2000'

UNION

SELECT Name(t)
FOR EACH Teacher t
WHERE Salary(t) > '2000';
```

The Object Algebra Equivalent of the Query is :

Researcher GEMMA (t) [t is an ELEMENT of <r>.Name AND <r>.Salary> '2000']<>

UNION

Teacher GEMMA (t) [t is an ELEMENT of <t>.Name AND <t>.Salary> '2000']<>

****

Object SQL Query :

SELECT Name(r)
FOR EACH Researcher r

UNION

SELECT Name(t)
FOR EACH Teacher t;

The Object Algebra Equivalent of the Query is :

Researcher GEMMA (t) [t is an ELEMENT of <r>.Name]<>

UNION

Teacher GEMMA (t) [t is an ELEMENT of <t>.Name]<>

****

Object SQL Query :

SELECT Name(r)
FOR EACH Researcher r

INTERSECTION

SELECT Name(t)
FOR EACH Teacher t;

The Object Algebra Equivalent of the Query is :

```
Researcher  GEMMA (t) [t is an ELEMENT of <r>.Name]<>

INTERSECT

Teacher  GEMMA (t) [t is an ELEMENT of <t>.Name]<>
```

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Tasmania'

UNION

SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Queensland';
```

The Object Algebra Equivalent of the Query is :

```
Person  GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Address= 'Tasmania' ]<>

UNION

Person  GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Address= 'Queensland' ]<>
```

****

Object SQL Query :

```
SELECT Name(p)
FOR EACH Person p
WHERE Address(p) = 'Tasmania'

INTERSECTION

SELECT Name(p)
FOR EACH Person p
WHERE City(p) = 'Hobart';
```

The Object Algebra Equivalent of the Query is :

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.Address= 'Tasmania']<>

INTERSECT

Person GEMMA (t) [t is an ELEMENT of <p>.Name AND <p>.City= 'Hobart']<>

References

Abiteboul, S. and Grumbach, S. 1990. "A Logic Based Language for Complex Objects", in *Advances in Database Programming Languages,* Addison Wesley (ACM Press) 1990, pp. 347 -374.

Ahad, R. and Dedo, D. 1992. "Open ODB from Hewlett-Packard : a commercial object-oriented database management system", *Journal of Object Oriented Programming,* February 1992, pp. 31 -35.

Ahmed et al, S. 1991. "A Comparison of Object-Oriented Database Management Systems for Engineering Applications", Research report R91-12, *Massachusetts Institute of Technology.*

Barry, D. K. 1991. "Perspectives on changes for ODBMSs", *Journal of Object Oriented Programming,* Vol 4, No. 2, July/August 1991, pp. 19 - 20.

Cain, L. 1989. *ANSI SQL parser*. Columbia University.

Date, C. J. 1990. *"An Introduction to Database Systems"*, 5th ed. Addison Wesley 1990.

Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lynbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A. and Shan, M. C. 1987. "Iris : An Object-Oriented databases Management System", *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, January 1987, pp.48 - 69.

Harris, G. and Duhl, J. 1990. "Object SQL" in *Object-Oriented DataBase with applications,* edited by Gupta and Horotwitz.

"OpenODB from Hewlett-Packard". Technical report.

"ITASCA Technical Summary", September 1991, Technical Report, Itasca Systems, Inc.

Kim, W. and Bancilhon, F. 1990a. "Object-Oriented Databases: Systems : In Transition", *SIGMOD RECORD,* Vol. 19, No. 4, December 1990, pp. 49 - 53.

Kim, W., Garza, J. F., Ballou, N. and Woelk, D. 1990b. "Object-Oriented databases: definitions and research directions", *IEEE Transaction on Knowledge and Data Engineering,* Vol. 2, No. 1, March 1990, pp. 109 - 123.

Kim, W. 1990c. "Object-Oriented databases: definitions and research directions", *IEEE Transaction on Knowledge and Data Engineering,* Vol. 2, No. 3, September 1990, pp. 327 - 340.

Kim, W. 1991. "Object-Oriented Database Systems: strengths and weaknesses", *Journal of Object-Oriented Programming,* July 1991,pp. 21 -30.

Lesk, M. E. and Schmidt, E., "LEX - A Lexical Analyser Generator", Bell Laboratories, Murray Hill, NJ.

Lyngbaek, P., Wilkinson, K. and Hasan, W. 1990. "The Iris Architecture and Implementation". *EEE Transaction on Knowledge and Data Engineering,* Vol. 2, No. 1, March 1990, pp. 63 - 75.

Lyngbaek, P. 1991. "OSQL: A Language for Object Databases", *Hewlett-packard Technical Report* HPL-DTD-91-4, 1991.

" Questions and Answers : Object-Oriented Databases", ONTOS, Inc. 1991.

Orenstein et al, J. 1992. "Query processing in the ObjectStore Database System", ACM SIGMOD, July 1992.

Unix System, Release 3.2, Programmer's Guide, Vol. 1, pp 5-1 - 5-21 and 6-1 6-57.A. T. Schreiner and h. George Friedman, Jr., " Introduction to Conpiler Construction with Unix", Prentice-Hall 1985.

Shaw, G. M. and Zdonok, S. B. 1989. "Object-Oriented Query Algebra",*Proceeding 2nd International Workshop on Database Programming Languages,* Oregon, 1989, pp 111 -120.

Straube, D. D. 1990a. "Queries and Query Processing in Object-Oriented Database Systems". *Technical Report* TR90-33, December 1990.

Straube, D. D. and Ozsu, M. T. 1990. "Queries and Query Processing in Object-Oriented Database Systems", *ACM Transactions on Information Systems*, Vol 8, No 4, October 1990, pp. 387 - 430.

Soloviev, V. 1992. "An Overview of Three Commercial Object-Oriented Database Management Systems : ONTOS, ObjectStore, and O2", *SIGMOD RECORD*, Vol. 21, No. 1, March 1992, pp 93 - 104.