THE IMPLEMENTATION OF A FUNCTIONAL QUERY LANGUAGE FRONT-END TO A RELATIONAL DATABASE SYSTEM

VOL I

HANIFA UNISA SHAH

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

June 1989

The University of Aston in Birmingham

THE IMPLEMENTATION OF A FUNCTIONAL QUERY LANGUAGE FRONT-END TO A RELATIONAL DATABASE SYSTEM

Hanifa Unisa Shah

Doctor of Philosophy

1989

Summary

Database systems have a user interface one of the components of which will normally be a query language which is based on a particular data model. Typically data models provide primitives to define, manipulate and query databases. Often these primitives are designed to form self-contained query languages. This thesis describes a prototype implementation of a system which allows users to specify queries against the database in a query language whose primitives are not those provided by the actual model on which the database system is based, but those provided by a different data model. The implementation chosen is the Functional Query Language Front End (FQLFE). This uses the Daplex functional data model and query language. Using FQLFE, users can specify the underlying database (based on the relational model) in terms of Daplex. Queries against this specified view can then be made in Daplex. FQLFE transforms these queries into the query language (Quel) of the underlying target database system (Ingres). The automation of part of the Daplex function definition phase is also described and its implementation discussed.

KEY WORDS: Functional Query Languages
Database Management Systems

To
my mother Zohra,
my husband Babar,
and the memory of my father Hanif.

ACKNOWLEDGEMENTS

This research and thesis have been completed with encouragement and assistance from various sources.

At Aston University - I am indebted to my supervisor David Avison for his constant encouragement and advice, often through difficult circumstances. His help and guidance have been invaluable. I would also like to thank, Dr Charles Hunt, Dr Ian Johnson, Dr Mike Parkes and Dr Nadia Bendjeddou for their helpful comments regarding this research and thesis.

At British Telecom - I am grateful to my manager Bilvir Chander for her support and encouragement and for allowing me the necessary flexibility in my work schedule to enable me to complete this research and thesis. My thanks to Robert Martin and Adrian Bell for suggesting the idea of loaning the necessary equipment from BT to complete this research and to Steve Bullock, Tony Goswell, and Mike Wild for providing this equipment. I am extremely grateful to Steve Woods for his assistance in setting up the hardware and software environment and getting it all to work successfully. My thanks also to my mentor at British Telecom, Bill McCarney for his encouragement.

At home - none of this would have been possible without the support of my mother, mother-in-law, brothers, sisters and their families. I would like to thank them for looking after my children on countless occasions so that I could continue with this research. Finally my thanks to my husband Babar, for his tireless support and constant encouragement. I am grateful to him, my daughter Amara and my son Haseeb, for putting up with the demands of this research and thesis on all of our lives.

LIST OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A database exists because it has been perceived to be necessary to record some information. This information is stored as data which will be used at a later stage when required, for, 'the purpose for all data-base structuring is retrieval: retrieval for output, retrieval for decision-making, retrieval for update' (Bachman, 1973, p59). A database system or database management system (DBMS) is a software system that provides the facilities to store and maintain the database and through which access to the stored data is gained.

One of the ways in which the information is used is through a query language interface. Chamberlin (1976, p49) refers to this as a 'stand-alone language in which an end-user interacts directly with the database management system'. This user interface is now considered an essential component of the database environment. It exists regardless of the underlying technology of the DBMS, though its exact form will vary from system to system. A significant problem of query language interfaces is that effective use depends to a large extent on the user understanding the data model on which it is based. The data model (Brodie, 1984) is a description of the data and its relationships.

There are other forms of user interface. A data sublanguage is a set of database operators intended to be embedded in a host programming language. A typical data sublanguage will be at a lower level of abstraction than a query language and therefore useful to a narrower range of users. Nevertheless, the same basic operators can serve both as a data sublanguage and as a query language. In many modern DBMS, the stand-alone query language is essentially the same as its data sublanguage embedded in a host language. For example in DB2 (Date, 1986), the language SQL is available in very similar forms both as an interactive interface and embedded in host programming languages, such as Cobol and PL/1. Another example is the Informix system (Informix, 1986), in which the language (again SQL) is available as an interactive query language and also embedded in the C programming language (ESQL).

In Date (1986) a data sublanguage is described as being a combination of at least two subordinate languages, a data definition language (DDL) and a data manipulation language (DML). A query language is described as a language in which high-level commands or statements may be issued to the DBMS, and the language SQL is specified as a typical example of a query language. However SQL is also described by Date as consisting of a DDL and a DML and therefore, can also be described as a data sublanguage. Michaels *et al.* (1976) also uses the terms data sublanguage and query language indistinguishably. This defines a data sublanguage as a language which offers capabilities for interrogating and updating databases.

Shneiderman (1980) uses the terms query language and DML synonymously to mean a data manipulation language for operating on data in the database. Others, for example Stonebraker and Rowe (1977, p128), state that all DML have an associated DDL to describe the data on which actions are performed and that: 'the DDL acts as a specification mechanism for the data in a database and cannot be avoided'.

In Paredaens (1987, p7), a DML is defined as: 'either a stand-alone programming language or is composed with several primitives which should be incorporated into a general purpose programming language ...'. The various terms DDL, DML and query language are described in Brodie, (1984, p20).

> 'Most existing database management systems provide a Data Definition Language (DDL) for defining schemas and subschemas, a Data Manipulation Language (DML) for writing database programs, and a query language (QL) for writing queries. The partitioning of these functions into three languages

is not necessary. Many database languages combine both query and manipulation. All three functions can be provided in one language.'

It is therefore apparent that the distinction between what is a query language and what is data sublanguage or DDL or DML is not well defined and the various terms query language, data sublanguage, DDL and DML are used to mean the language component available for accessing the data in the database. This is appropriate to a certain extent since they all have in common the fact that they are concerned with database objects and database operations. In this thesis, a query language is defined as consisting of both a DDL and DML.

In general, DBMS do not provide more than one query language interface to the database, though they may provide a forms-based or graphical interface as well as a host language interface. Current DBMS, for example Informix, Ingres and DB2, provide more than one means of accessing information in the database, but few DBMS have more than one of each of the different types of interface, while none provide an alternative query language based on a different data model than that underlying the DBMS. If users of a particular DBMS wish to use a query language, they are limited to using this single query language provided with that DBMS. Unfortunately there is not a standard query language for each data model. Different DBMS based on the same data model often use a different query language.

It would be less restrictive if DBMS had available more than one query language interface which used different data models, particularly if the additional query language could be ported to a number of DBMS. Such a capability could be provided by means of a front-end to deal with queries expressed in the alternative query language (see figure 1.1). A front-end is a software system that is built on top of an existing system. It maps user queries into the system target language using the relevant information about the user's view. The target language would be the query language of the underlying DBMS. This would allow users to view their data in different ways and also allow them to use different DBMS with a reduced learning cycle.

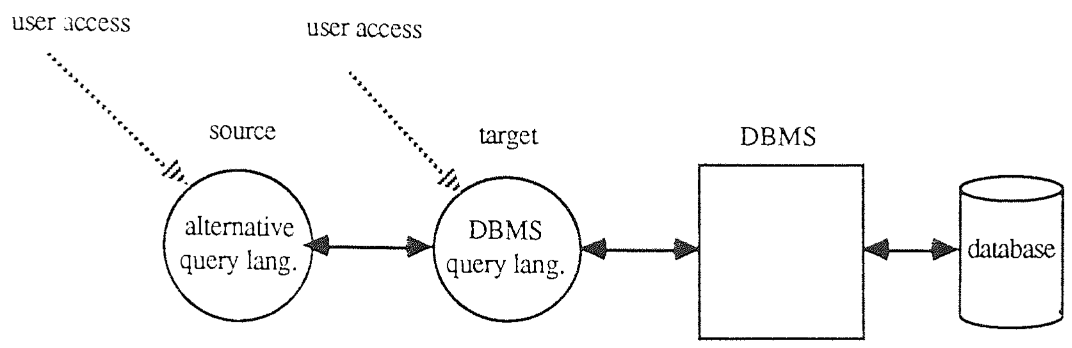


Figure 1.1: Front-end System and DBMS Interaction

This thesis presents and discusses an implementation of a query language front-end based on a data model and query language that is different to the data model and query language of the underlying DBMS. The provision of another query language interface to the same database means that the user's view is independent of the actual structure of the underlying database.

In this way more than one view of the database is supported. This is of significance since expanded use of DBMS technology has generated a considerable growth of the user community and of the variety of databases. A framework for multiple-view support is discussed in Klug & Tsichritzis (1977) and Tsichritzis & Klug (1978). Fundamental to this framework is the idea that data models are conceptual tools and that users should not be forced to visualise their data in one way only. However, there is considerable investment by manufacturers and users in current systems, and it is more realistic for future systems to support the traditional approaches as well as new approaches.

Different data models are supported by DBMS due to reasons of implementation history and because of the suitability of different data models to specific applications. There are therefore a number of query languages available. By allowing the same DBMS to support more than one query language interface, this valuable diversity is not lost (Manola & Pirotte, 1983).

A number of recommendations for a unified approach to query languages have been made by the British Computer Society (BCS, 1981). This approach should 'make access to computer held data easier and more consistent'. This report specifies the need for a view of data that is understandable to users and suitable to a variety of applications. More particularly, it argues that there should be a set of facilities for users to be able to express queries in a variety of models and styles of dialogue. One form of dialogue for a query language is not advocated, but different query language interfaces should be supported within the same system.

The research presented in this thesis goes some way to fulfilling the requirements of a uniform approach to query languages since it provides an additional query language interface to an existing DBMS. The front-end system presented in this thesis is highly portable due to the well-defined interface between it and the DBMS (see figure 1.2). It can be adapted to a different DBMS relatively easily. This is aided by its implementation language and environment being common to a number of computer systems.

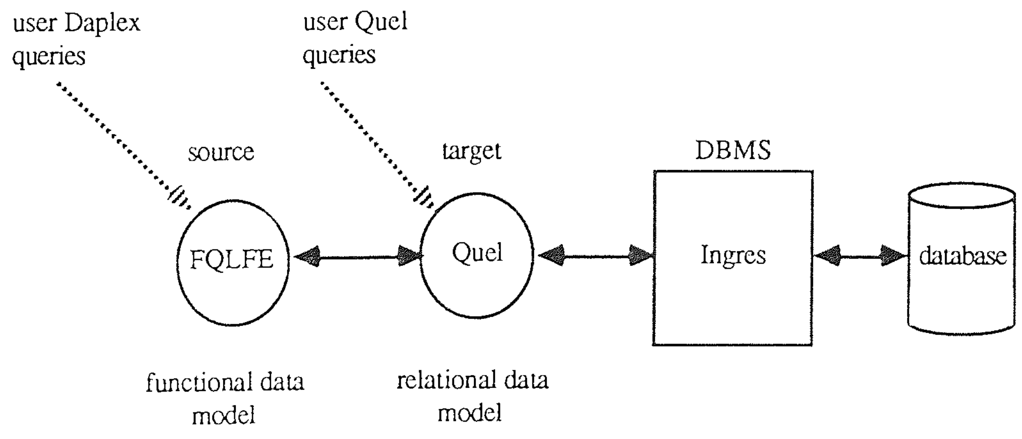


Figure 1.2: FQLFE and DBMS Interface

The software system designed during this research is FQLFE (Functional Query Language Front-End). It is built on the relational database management system Ingres, which has as its query language Quel (described in section 2.3.5). FQLFE accepts user queries expressed in the functional query language Daplex (described in section 2.3.12), and addresses databases based on the functional data model. The system transforms the users' Daplex-expressed queries and maps these into Quel queries. These are then executed by the underlying DBMS. The reasons for the choice of query languages and the data models used as a basis for this research are detailed in section 2.4.

The objectives of this research were to show that a system could be implemented that allows the database to be queried in a language different to the one that is available in the DBMS and that this query language could be based on a data model that is different to the data model of the actual DBMS. The FQLFE system presents an interface to the user that consists of a functional query language to query databases based on the functional data model. The database is based on the relational data model and is queried by a relational query language.

Chapter 2 of this thesis examines a number of query languages which typify particular approaches and which may be suitable for this implementation. Chapter 3 examines the chosen implementation environment in more detail. The functional data model and query language Daplex are discussed. The relational DBMS Ingres and its query language Quel are also described. Chapter 4 presents details of the FQLFE system which is a front-end based on the query language Daplex to the relational DBMS Ingres. Chapter 5 presents an application to illustrate the use of the FQLFE system. It provides examples of queries that can be made in the Daplex language against the Ingres database and it gives the equivalent Quel transformations of these. Chapter 6 gives a summary of the research, the conclusions reached and the directions for further research. Volume II consists of the Appendices. Appendix A gives the syntax of the Daplex query language. Appendices B and C give the lex and yacc specifications used. Appendix D consists of the program listings for FQLFE. Appendix E contains the results produced by the tests discussed in chapter 5 of the thesis.

CHAPTER 2

QUERY LANGUAGES

2.1 INTRODUCTION

An outline architecture for a database system proposed in Tsichritzis & Klug (1978) is divided into three levels as follows:

- *the conceptual level* - this is the level concerned with the expression of the total operational data required, that is, the concepts represented by the data;
- *the internal level* - this is the level concerned with how the data is actually stored, that is the physical representation of the data;
- *the external level* - this is the level concerned with the way the data is viewed by individual users.

It is this external level that is of most significance for this research, since this is the individual user level. At this level each user will have available a language. The different users are:

- *the database administrator* - who will have available the languages available to the other users and possibly a system level language
- *the applications programmers* - who will have available either conventional programming languages, such as Cobol, or a proprietary programming language specific to the system, for example, dBaseIII
- *the end-users* - who will have available either a query language or a special purpose language, such as, a forms or menu based language, or possibly both.

All such languages will include a data sublanguage, that is a subset of the total language that is concerned specifically with database objects and operations. The form that this data sublanguage appears in will vary both from DBMS to DBMS and also within a particular DBMS. For example in the Informix DBMS the data sublanguages exist in the following forms:

- in the query language SQL,
- embedded in the host programming language C,
- as a forms front-end, and
- as a report writer front-end.

Other ways in which data access is available within DBMS are:

- natural languages,
- graphical interfaces, and
- menu-driven interfaces.

Particular DBMS will have some combination of data access interfaces. This does not mean that all of these are always provided. As shown in chapter 1, in some cases the DDL and the DML form the query language and sometimes the DML alone is referred to as the query language. Often this query language is embedded in a programming language and the forms, menu, and graphical interfaces also translate internally into some form of the query language. Natural language interfaces are outside the scope of this thesis.

In the next section a number of criteria for comparing query languages are discussed and in section 2.4 query languages are discussed on the basis of this set of criteria.

2.2 QUERY LANGUAGE FEATURES

Two query languages were selected in this research. The first is the target language, that is the language which is normally used for the DBMS. The second is the front-end language, that is the alternative which a user may choose for the DBMS. The features chosen for comparing query languages are relevant to both choices, though their relative importance and interpretation will differ. The criteria are: the data model upon which the query language is based, its simplicity, completeness, level of procedurality, the support it provides for higher level languages, its ease of extension and availability.

2.2.1 Data Models

Data models provide the conceptual tool for thinking about data-intensive applications and they provide a formal basis for the tools and techniques used in developing and using information systems. The data structures upon which the database system is based influences the query language used for accessing the database. The available data structures and the allowable operations on them are features of the underlying data model.

A paper by Manola & Pirotte (1983, p68) defines the relationship between query languages and data models as follows:

> 'A query language is based on a data model when the query language addresses the data structures of the data model and the semantics of its expressions can be expressed with operations of the data model'.

Brodie (1984, p20), describes the relationship between data models and the languages supported by them as follows:

'Tools associated with data models are languages for defining, manipulating, querying and supporting the evolution of databases.'

From the users' point of view, the relationships being modelled by a DBMS must be understood regardless of the query language or data model. It should be noted that understanding the schema should not be difficult if it accurately models reality.

Ullman (1982) defines data models as consisting of two elements:

- A mathematical notation for expressing data and relationships.
- Operations on the data that serve to express queries and other manipulations of the data.

The first of these basic components is a set of rules to describe the structure and meaning of data in a database and the second consists of the atomic operations that may be performed on the data in that database. The rules express the static properties of a data model and correspond to what is usually called a data definition language. They define the structures that are allowable within the data model as a set of schemas. Different occurrences of the database can correspond to the schema. A database state corresponds to a particular database occurrence. The set of operations of a data model, called a data language, defines the allowable actions that can be performed on a database occurrence. That is, each operation maps one database state to another database state.

Data models enable us to capture, partially at least, the meaning of the data as related to the complete meaning of the world. The data represented in the data model may have meanings which are unknown or irrelevant in terms of modelling, it is sufficient that the meaning captured by the data model should be adequate for the purpose required. The data model acts as a basis for high-level query languages for data retrieval and data manipulation.

Operations of a data model (referred to as a data language in Tsichritzis & Lochovsky, 1983), transform a database from one state to another. When operations are performed on a database, they tend to be focussed on one small area of the database, and this implies a selection process. Operations usually consist of an action and a selection. The action specifies what is to be done, and the selection specifies the part of the database to which the action is to be applied. Actions are drawn from: retrieve, insert, modify and delete operations.

The fundamental aim in data modelling is to organise data so that it represents the real world situation as closely as possible, yet can still be represented by computers. The problem is that these are conflicting requirements.

By combining different possible components for structures and operations, many different data models are possible. However, not all of these would be useful or practical. It is true to say that very few data models have received wide acceptance and use. There are many proposed data models, however not all of these specify consistent operations for manipulating the data structures provided by the data model.

The earliest data models to be described for databases are the hierarchical, network, and relational data models. These three are considered to be the 'basic' data models and are discussed in most texts on databases, for example, Tsichritzis & Lochovsky (1978), Ullman (1982), Date (1977 and 1986), and Pratt & Adamski (1987). These three data models are often used as the yardstick to compare other data models and their capabilities.

2.2.1.1 Relational Model

The relational data model is based on the concept of mathematical relations and was first described by Codd (1970). Examples of systems based on this model are

Ingres (Held *et al.*, 1975, Stonebraker *et al.*, 1976, and Date, 1987); Informix (Informix, 1986) and Oracle (Oracle, 1986). In this model a relation is a collection of instances (described in rows, known as tuples) of a record type in which the ordering of the instances and the ordering of the fields within the record type are not significant. The relationships between relations are not specified explicitly. Relations describe entities and relationships between entities. The entire information content of the database is represented as explicit data values in column positions within rows of tables. Inter-relationships which are not represented as tuples can be established at access time using the relational language interface. In this way the user is not restricted to the predefined relationships represented in the schema - this is one of the major differences between this model and the hierarchical and network models. Further discussion of the relational model can be found in the texts by Date (1986), Tsichritzis & Lochovsky (1983), Ullman (1982) and Reiter (1984).

In this model there is a uniformity of data representation, this in turn leads to a corresponding uniformity in the set of operations (Date, 1977 and 1986). Most of the work in query languages has been based on the relational model, some of which is described by Chamberlin (1976), Codd (1971 and 1972), Stonebraker *et al.* (1976), Astrahan & Chamberlin (1975), Mohan (1978) and Zloof (1977).

Data manipulation languages for the relational model are usually derivatives of relational calculus or relational algebra (Codd 1970). Languages for this model tend to be relatively non-procedural (section 2.2.4). This means that operations in these languages are specified in terms of names and values only and users do not need to know about the physical representation of the data or the paths to be followed to arrive at the data.

2.2.1.2 Network and Hierarchical

The Codasyl Data Base Task Group report (Codasyl, 1971) is the most comprehensive specification of a network data model. The terms are often used synonymously, though the Codasyl model can only be considered to be one implementation of a network model. Examples of systems based on these proposals are: IDS and IDMS. The hierarchical model is a subset of the network model.

The network and hierarchical models use the concept of a record as a collection of named fields to represent each individual object in the application environment. In the network model the set mechanism establishes a one-to-many association between any owner record and its member records. A group of these sets can form a network of relationships. In the hierarchical model (Date, 1986, Tsichritzis & Lochovsky, 1978, Pratt & Adamski, 1987) there is a tree-like set of one-to-many relationships, in which each record occurs at a single specified level of the hierarchy. In these two models users are restricted to pre-defined relationships. This means that databases based on these models are not very dynamic, that is, they are not readily changeable in terms of structure and specification of relationships.

The data manipulation languages for systems based on these languages tend to be navigational, that is, the user must access the database by explicit traversal through the hierarchy or network. This contrasts with the relational model where access to the data is by specifying the properties of the data of interest rather than the route that is to be followed to obtain it.

2.2.1.3 Other data models

Since the basic data models have been in existence, many other data models have been specified. They are, at least to a certain extent, extensions of the basic data models.

The *role model* defined by Bachman & Dayal (1977) has been developed as an extension of the network data model. It includes the notion of a role. An object may play many different roles in an application and may have different properties for each role. For example, a person can play the role of employee, manager, and so on. The role model reduces the redundancy of considering each role as a separate object. Note however that its language interface is still a navigational one and therefore the operations available on its basic data structures are still at a relatively low level.

In the *structural model* (Weiderhold & El-Masri, 1979), relations in the relational model are restricted to specific types. Relations specifying a set of independent objects are entity relations, those specifying one-to-one correspondence between names are lexicons, and those representing many-to-many relationships are associations. These restrictions guide database design choices and encourage precision in using the relational data model. In this model no distinction is forced between objects and relationships, the resulting schemas are therefore relational. As this model is based on the relational model, languages for this model are relatively nonprocedural.

Another data model is the *entity-relationship model* (Chen, 1976), which combines features of the network and relational models. It makes a clear distinction between objects and relationships. Applications are represented as networks in which entities are nodes and relationships are edges. The entity-relationship model provides a means for representing relationships, rather than a small set of relationship concepts. It has become a popular model for high-level database design due to its economy of concepts, and also because it is generally accepted that entities and relationships are natural modelling concepts. There are no current DBMS for which the entity-relationship model is the underlying data model, databases designed using this are mapped into DBMS based on the other data models.

A group of data models called *functional data models* have also been proposed. These are usually classified as irreducible, since the objective is to represent information as atomic facts rather than as complex groups of facts. These atomic facts cannot be decomposed further (Brodie, 1984). Atomic facts simplify the update operation since each fact can be updated independently. Further, they increase modelling precision, since these atomic facts can be combined in any appropriate way to form any reasonable higher level concept. In this way no fixed structure is imposed on all facts.

The functional data models combine aspects of the relational data model with functional programming. That is the database is regarded as a collection of functions over data types, such as employee and department, and basic types, such as character and integer. The relationships between objects are represented by functions. One of the functional data models is the functional dependency model (Housel *et al* .1979). In these proposals a database is described as consisting of sets of values with functions between them. There are two types of sets. Simple sets correspond to integers, strings, etc. (that is, basic data types). Tuple sets correspond to many-to-many relationships among sets. Operations are provided for retrieval, update, insertion and deletion of occurrences in functions and sets.

Another of the functional data models is built into the query language FQL (Buneman & Frankel 1979 and Buneman *et al.* 1982). In this model the database is viewed as a collection of functions over various data types. Five operators are provided which combine functions to form new functions, and operators also enable arithmetic and boolean operations in query formulation. Its notation is derived from that specified by Backus for functional programming (Backus, 1978).

The most widely discussed and researched of the functional data models is that defined by Shipman (1981). It incorporates a high-level, integrated language for data definition and manipulation called Daplex. In this model the basic concepts are entities and functions. The database is modelled as a set of functions mapping entities to entities. Functions may have zero or more arguments. Entity types are defined by functions with no arguments. Entity attributes and relationships among entities are defined by functions with arguments. The query language interface Daplex has statements for iteration through sets and statements for updating, and it also has a special operation for incorporating user-defined functions in the schema. The Daplex query language is described in detail in chapter 3.

A number of conclusions can be drawn about data models in relation to query languages. Firstly, query languages are largely influenced by the data model upon which they are based. The data structures upon which it operates are defined by this underlying data model. The data models for which there are well defined consistent operations are the 'basic' data models, that is, relational, hierarchical and network. Of the other data models the functional data model as defined by Shipman (1981) also has a well defined and consistent set of operations. The relational data model and its query languages are well established in modern DBMS, since the concepts on which they are based are relatively easy to understand. Shipman's functional data model is also conceptually natural, since it is based on the concepts of entities and functions.

2.2.2 Simplicity

This is usually an important factor for expressing queries. This feature is discussed in Shneiderman (1980), which recommends that in the query language there should be a small number of concepts required to get started and simple operations in the language should be expressed in a simple way. It also suggests that the language syntax should be relatively simple, even for complex operations. Further, there should be consistency in the language and various operators should have consistent semantics in all contexts. The concept of simplicity also means that the language should have flexibility and should model reality in a way that is meaningful to the user. Ideally, the structure and syntax of such a language should be uniform.

2.2.3 Completeness

This term was first defined by Codd (1972). It has usually been considered a fundamental requirement for relational languages. Codd defined a language called relational calculus based on the first order calculus. A language is said to be relationally complete if it has at least the same expressive power as relational calculus. Thus, for any expression, there is a relational calculus expression equivalent to it. It is the measure of the selective power of a query language. However it should be noted that there are some features that a good query language should possess that are not covered by the definition of relational completeness, for instance, aggregate functions and arithmetic capability. This additional power will mean that for a very large class of queries, the user will never need to use loops or branching to extract the data required. Thus some popular relational languages are 'more than complete', that is their retrieval power is superior to that of the relational calculus. However relational completeness provides a useful means of comparing query languages.

The concept of relational completeness has rarely been extended to languages based on any data models other than the relational data model, Though it has been applied to the network model (Tsichritzis & Lochovsky, 1978) and to the entity-relationship model (Atzeni & Chen, 1983). It has been achieved by redefining the definition for relational completeness, since it can only naturally be applied to relational algebra or calculus based languages. Relational completeness can only really be used as a comparison mechanism for languages that are based on relational concepts. Some researchers, for example Reisner (1981), Cuff (1982) and Shneiderman (1980), feel that this is a rather primitive measuring rod for assessing the capabilities of a query language and do not consider it to be essential, because:

- many queries that can be written with a relationally complete language are extremely difficult to compose or comprehend, and
- many common, useful, simple to understand and potentially easy to express queries are outside the bounds of relational completeness.

2.2.4 Nonprocedurality

Procedural languages require the user to specify the process or path to be followed in order to obtain the result. Nonprocedural languages (also called specification languages) are those which specify the result required. These are usually viewed as being 'high-level', and queries expressed in them are normally shorter in length than those expressed in procedural languages. There has been much debate about the relative merits of these two approaches, especially in terms of writing queries and understanding queries written by someone else (Stonebraker & Rowe, 1977).

Set level languages are often referred to as non-procedural. These specify what is required not how to retrieve it. Relational systems are often referred to as 'automatic navigation' systems, since the process of 'navigating' (that is, specifying the path to be followed to answer a query) around the database is done automatically. The concepts of procedurality and nonprocedurality are often assumed to be absolutes. This is not the case, they are relative to one another and can be considered to be at different levels of abstraction. Nonprocedural languages are at a higher level of abstraction than relatively procedural languages, and are considered to be more powerful and easier to use.

2.2.5 Support for higher level languages

For particular users or applications, some of the languages may be suitable in their basic form. Other users may require their own special purpose languages, involving, for instance, terminology specific to their own application area. The specified languages are examined to see if they provide a common core of features that will be required by other higher level languages. The system would then map these higher level languages into the specified target language.

2.2.6 Ease of extension

Although a language may be relationally complete, it may still be inadequate for some queries. The user may need to retrieve values that are not necessarily stored in the database, but may be computed in some way from the information in the database. For this reason the basic retrieval power of the language is enhanced by being able to be easily extended to include useful standard functions.

2.2.7 Format

Another feature that varies from query language to query language is the format. Some query languages are keyword oriented, for example, Quel (see section 2.3.5 and SQL (see section 2.3.6). Keyword-oriented languages are those which query the database using a syntax consisting of command types expressed as keywords and names and values of database objects. These base their structure on traditional programming languages. Other query languages are based on a two-dimensional notation, for example, Cupid (see section 2.3.9) and QBE (see section 2.3.7). In these languages positioning is critical and very few keywords are used. The implementations of these languages may require a graphics support system. The advantage of keyword-based languages is that keywords may help in the processes of learning and composing queries in the query language. The two-dimensional languages have an advantage that confusion might be reduced by using positional notation or special shapes to specify queries.

2.2.8 Availability

Many query languages have been proposed, but for various reasons they have never been implemented. This may be due to the fact that the data models upon which they are based have never been implemented or because no significant advantage has been foreseen by researchers or commercial organisations to implement them. In selecting a suitable target language, availability is an important factor that must be considered. If the front-end is to be widely used, its target language and DBMS should be widely available, otherwise its usefulness will be very limited.

2.3 EXAMPLE QUERY LANGUAGE INTERFACES

In this section, a number of query language interfaces to DBMS are described and discussed in terms of the features which have been specified in the previous section. Sample queries are given, based on the same database wherever possible. The structure of the database, expressed in terms of relations, is given below. This is similar to an example used in several texts, for example, Lacroix & Pirotte (1978) and Date (1986):

```
suppliers (snumber, sname, status, city)

parts (pnumber, pname, color, weight, city)

shipments (snumber, pnumber, qty).
```

2.3.1 DL/1 (Data Language/1)

This is the data sublanguage that is available for accessing the IBM hierarchical DBMS, IMS (Information Management System). This is included in the discussion because prior to the wide acceptance of relational databases it was the most widely used DBMS. The DL/1 language is invoked by subroutine calls from the host language of the system. This could be PL/1, Cobol, or Assembler.

The hierarchical model supports a tree structure, in which the tree is inverted. This tree consists of nodes connected by branches. The root node is at the top and its descendants are below it. A parent node appears immediately above its children. Each node contains one or more fields or attributes. The main feature of this model is that each node is accessed through its parent. That is, access to the data is along a hierarchical path from top to bottom.

The DL/1 terminology refers to a node as a segment. This is equivalent to the relational term table and the Codasyl term record. One tree structure occurrence of the root segment and all its descendants is called a physical database record (PDBR), while the collection of all physical database records for a particular tree structure is called a physical database. In DL/1, a number of physical databases can exist. Each one will be defined separately in a single database definition. In this definition (called a DBD) all segments, all fields within each segment, and all hierarchical relationships between segments are defined. The definition part of DL/1 involves defining the DBD. There are a number of other definitions required and these are described in detail in Date (1977) , Pratt & Adamski (1987) and Taylor & Frank (1976).

In DL/1 an application language (for example, Cobol) interacts with the databases by using DL/1 DML requests. A request consists of a `CALL` statement followed by a parameter list. A summary of DL/1 function codes using a simplified syntax is given.

- `GET UNIQUE (GU)` - Direct retrieval of a segment occurrence that satisfies the given argument.
- `GET NEXT (GN)` - Sequential retrieval of the next segment occurrence.

- GET NEXT WITHIN PARENT (GNP) - Sequential retrieval of the next segment occurrence under current parent.
- GET HOLD UNIQUE (GHU) - Same as GNP but allows subsequent DLET and REPL.
- GET HOLD NEXT (GHN) - Same as GN but allows subsequent DLET and REPL.
- GET HOLD NEXT WITHIN PARENT (GHNP) - Same as GNP but allows subsequent DLET and REPL
- DELETE (DLET) - Delete an existing segment occurrence.
- RELACE (REPL) - Replace an existing segment occurrence
- INSERT (ISRT) - Add a new segment occurrence.

The structure of a query if expressed in DL/1 would be as follows.

- Q1. Get supplier numbers for suppliers who supply part P2.

This query involves the following stages:

```
                Get next parts where pnumber  = P2
    REPEAT:     Get next suppliers for this parts
                suppliers found? No - exit.
                Yes - print snumber
                REPEAT.
```

For application systems that have fixed, predefined relationships, DL/1 is a logical means of expressing such systems. However many data relationships are not hierarchical. A problem with DL/1 is the difficulty of representing many-to-many relationships. Further, there is very little data independence and powerful hierarchical databases are difficult to understand and to use, and they require a significant amount of expertise. Additionally, in DL/1, information requirements that do not follow the natural hierarchical path may be time consuming to address.

2.3.2 Codasyl (DML)

Codasyl databases are constructed of records, pointers to records and files of records which contain pointers to other records. The DML has the power to follow pointers and retrieve records. This type of interface is known as a record-at-a-time interface. These were the earliest 'query languages' and were based on 'navigating' around the database. The fundamental concept underlying this DML is the idea of currency. This currency is the value of certain system created and maintained pointers. The database is traversed record-at-a-time until the sort of record that the user wishes to retrieve is pointed to by the currency indicators. This record is brought into the work area and the host language carries out the relevant processing on it. To be able to use such a language one must know the details of the database and its structure and physical implementation. The DML is dependent on a host language for all iterative scanning and searching, and also for evaluating all of the records.

A record type is the basic structure while a record occurrence is a specific example of this structure. Relationships are maintained in Codasyl systems by means of a construction called a set. A set type is a one-to-many association between record types. The record type that represents the 'one' part of the association is called the owner record type, and the record type that represents the 'many' part of the association is called the member record type. An occurrence of the set type is a single occurrence of the owner record type, together with the many occurrences of the member record type that are related to it.

The overall logical structure of the database is represented by the schema. It is set up on the computer using a language called the schema data definition language (schema DDL). An individual user's view of the database is represented by the subschema. This is specified on the computer using the subschema DDL. Programs access the database subschemas. Therefore the various forms of the subschema DDL are tailored to different languages. Programs accessing the database do so through the normal commands present in the language in which the program is written and through additional database accessing commands. These commands form the DML. The currency indicators are:

- *current of run unit* - the run unit is the program. There is only one current of run unit, this will be the last occurrence of any type of record that was retrieved or saved.
- *current of record type* - there is one of these for each record type in the subschema. The current of a record type will be the last occurrence of that record type that was retrieved or saved.
- *current of set type* - there is one of these for each set type in the subschema. The current of a given set type will be the set occurrence most recently accessed.
- *current of realm* - there is one of these for each realm (or area) in the subschema. The current of a given realm (a subset of the database) will be the last record of any type that was found or stored in that realm. This conceptual pointer is rarely used.

The DML commands available are as follows, though the exact syntax may vary depending on the host language.

- `FIND` - locates a record subject to some conditions.
- `GET` - retrieves the contents of the record identified as current of run unit.
- `STORE` - creates a new record occurrence.

- `MODIFY` - updates the current of run unit record.
- `ERASE` - disconnects the current of run unit from occurrences of any set in which it is a member and deletes the current of run unit, provided that the current of run unit does not own any member occurrences in any set.
- `ERASE ALL` - disconnects the current of run unit from occurrences of any set in which it is a member and deletes the current of run unit and any members of set occurrences owned by it.
- `CONNECT` - connects the current of run unit into a set occurrence.
- `DISCONNECT` - disconnects the current of run unit from a set occurrence.

The network approach is similar to the hierarchical approach, where the data is represented by records and links. A network, however, is a more general structure than a hierarchy, since a given record occurrence may have any number of immediate superiors, and not just one, as is the case with a hierarchy. Record types are used to represent the suppliers and parts and also the association between suppliers and parts. The record representing the association is called a connector, and an occurrence of this contains data describing the association. A query using the Codasyl DML would involve the stages shown:

- Q1. Get supplier numbers for suppliers who supply part P2.

```
                get next parts where pnumber = P2
REPEAT:         get next connector for this parts
                connector found? No - exit
                yes - get owner suppliers for this
                      connector
                print snumber
                REPEAT
```

• Q2. Find the part numbers for parts supplied by supplier S2.

```
            get next suppliers where snumber = S2
REPEAT:     get next connector for this suppliers
            connector found? No -exit
            yes - get owner suppliers for this
                connector
            print pnumber
            REPEAT
```

When compared to the relational model based query languages, the Codasyl DML loses out in terms of simplicity. This is due to the inherent problems of the data model upon which it is based. The necessity to continually navigate the database in an appropriate way for any database access adds a level of complexity not present in the relational model. In addition, the greater fragmentation of information is characteristic of the network model. It is a highly procedural language in that the path that must be followed to satisfy the query has to be specified explicitly.

2.3.3 Relational Algebra

This is a set of high-level operations on relations. This set of operations was first defined by Codd (1972), and was shown to be relationally complete. The relational algebra consists of two groups of operations. These are:

- *the set operations* - `union`, `intersection`, `difference` and a form of `Cartesian product`, and
- *special relational operations* - `selection`, `projection`, `join` and `division`.

These operators are all used in the retrieval of data, while only `union` and `difference` are used in update operations. Each operation of the relational algebra takes either one or two relations as its operands, and produces a new relation as its result. It is one of the strengths of the relational approach that languages such as relational algebra, which are relatively simple and yet very powerful, can be readily defined for it.

SELECT is an operator for constructing a subset of tuples within a relation for which a specified predicate is satisfied. The predicate is expressed as a boolean combination of terms. Each of these terms is a simple comparison that evaluates to true or false by inspecting that tuple in isolation. PROJECT is an operator for constructing a subset obtained by selecting specified attributes and disregarding others. It also removes duplicate tuples. The JOIN operator is a join based on equality values in the common domain. In addition to equality, joins may be defined in terms of the other comparison operators, such as not equal to, greater than, less than greater than or equal to and less than or equal to. In all these cases it is essential that the attributes being joined are based on the same domain. DIVISION takes two relations, one binary and one unary, and builds a relation consisting of all values of one attribute of the binary relation that match in the other attribute all values in the unary relation.

Having introduced the basic operations available in relational algebra, some example queries are shown:

• Q1. Get supplier numbers for suppliers who supply part P2.

This query can either be expressed as two separate statements for clarity or it can be expressed in a nested relational algebra expression.

```
SELECT shipments WHERE pnumber = 'P2' GIVING temp

PROJECT temp OVER snumber GIVING result
```

or

```
PROJECT(SELECT shipments WHERE pnumber = 'S2')
          OVER snumber GIVING result
```

• Q2. Get supplier numbers for suppliers who supply at least one red part.

Again this query could be divided into several statements or expressed as follows.

```
PROJECT(JOIN(SELECT parts WHERE color = 'RED')
          AND shipments OVER pnumber)
                    OVER snumber
GIVING result
```

• Q3. Get supplier names for suppliers who supply all parts.

```
PROJECT(JOIN (DIVIDE (PROJECT shipments OVER snumber,
                              pnumber)
                    AND shipments OVER pnumber)
                              OVER snumber) OVER sname
                                        GIVING result
```

• Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation parts.

```
parts UNION {('P7', 'WASHER', 'GREY', 2, 'ATHENS')}
                              GIVING parts
```

• Q5. Delete supplier S1.

```
suppliers MINUS {('S1', 'SMITH', 20, 'LONDON')} GIVING
                    suppliers
```

Relational algebra is relatively powerful and simple, although it is relatively procedural rather than nonprocedural, since the order in which queries are evaluated has to be specified. It is a relationally complete language, but is not readily extendible to include other operations such as aggregation. Because it is nonprocedural and not readily extendible, it is unsuitable as a common target language for higher level language translation. Although it is keyword-based, and relatively simple, its syntax is not very user-friendly. Furthermore, its availability is limited.

2.3.4 Relational Calculus

The relational data language Alpha (Codd, 1971) has as its theoretical foundation the relational calculus. The result of any query on a relational database may be considered as a relation. The relational calculus is a notation for expressing the definition of some new relation in terms of some given collection of relations. In this language, the relations that already exist as part of the database are used to define the relation that is to be the result of the query, for example,

```
{(shipments.pnumber, suppliers.city) :
                    suppliers.snumber = shipments.snumber}
```

is an expression of the query 'find the supplier city and part number of each shipped part'. The braces `{ }` specify that the expression is a relation definition. The colon stands for 'such that', and the term preceding the colon represents a typical tuple of the set, while the term after the colon is the predicate or condition that defines the property of the target relation. The result of the complete expression is the set of all `(pnumber, city)` pairs such that the `pnumber` value is from a `shipment` tuple and the `city` value comes from a `suppliers` tuple and the `snumber` values in these tuples are equal.

Other notation used in the following examples are the symbols $\wedge$, $\vee$, $\neg$, ( ), $\forall$, to represent and, or, not, forced order of evaluation and the universal quantifier respectively. The symbol $\exists$ represents existential quantification and is read as, "there exists". The same example queries used for illustrating the relational algebra are now given in this relational calculus based language.

• Q1. Get supplier numbers for suppliers who supply part P2.

```
GET  W(shipments.snumber) : shipments.pnumber = 'P2'
```

• Q2. Get supplier numbers for suppliers who supply at least one red part.

```
RANGE parts px

      GETW(shipments.snumber):∃px(px.pnumber =

            shipment.pnumber Λ px.color='RED')
```

• Q3. Get supplier names for suppliers who supply all parts.

```
RANGE parts px

RANGE suppliers s

RANGE shipments spx

     GET W(s.sname):Vpx∃spx(
     spx.snumber=s.snumberΛspx.pnumber=px.pnumber)
```

• Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation parts.

```
W.pnumber = 'P7'

W.pname = 'WASHER'

W.color =  'GREY'

W.weight = 2

W.city = 'ATHENS'

PUT W(parts)
```

• Q5. Delete supplier S1.

```
DELETE (suppliers) : suppliers.snumber = 'S1'
```

Relational calculus differs from relational algebra in that whereas the relational algebra provides a collection of explicit operators - `join, union, projection` and so on - that can actually be used to build a required relation from the given relations in the database, the calculus simply provides a notation for specifying a definition of that required relation in terms of those given relations.

The main feature of relational calculus is that it is nonprocedural, unlike relational algebra. It specifies the resultant relation, not the way in which it is to be obtained. However, it should be noted that the algebra and calculus are precisely equivalent to one another. This means that every relational algebra expression has an equivalent relational calculus expression and vice versa. Although the notation is unfamiliar at first, the data language Alpha is relatively simple: simple operations can be expressed simply, and the complexity of statements in this language is in direct proportion to the complexity of the operation that the user is attempting to perform. As discussed in the previous section, this feature is particularly important when assessing query languages.

This language has all the advantages of relationally complete languages. The language lends itself to extension, its power can be extended indefinitely and simply by providing library functions. Although the language may not be suitable in this form for many users, it possesses a common core of features which are going to be required in one way or another in all query languages. It would therefore be reasonable to use it as the target language for high level language translators. Although Alpha itself is not available commercially, other languages which have relational calculus as a basis have been developed (for example, Quel), which are at a higher level of abstraction than the Alpha language considered here, and could be suitable as a target language.

2.3.5 Quel

This is the query language for the Ingres DBMS. A brief description of Quel is given here as it is described in more detail in chapter 3. Quel is available both as an interactive query language and also as a database programming language embedded within a variety of host languages. Quel consists of both DDL and DML functions. The data definition part of the language consists of facilities to create databases, relations within databases and also facilities to remove relations, create indexes and also to define views. The basic DML statements available are, `retrieve`, `replace`, `delete` and `append`. Some example queries in the Quel language are now given.

- Q1. Get supplier numbers for suppliers who supply part P2.

```
RANGE OF sp IS shipments

RETRIEVE (sp.snumber)

WHERE sp.pnumber = "P2"
```

- Q2. Get supplier numbers for suppliers who supply at least one red part.

```
RANGE OF sp IS shipments

RANGE OF p IS parts

RETRIEVE UNIQUE (sp.snumber)

      WHERE sp.pnumber = p.pnumber

            AND p.color = "RED"
```

- Q3. Get supplier names for suppliers who supply all parts.

```
RANGE OF s IS suppliers

RANGE OF p IS parts

RANGE OF sp IS shipments

RETRIEVE (s.sname) WHERE ANY (p.pnumber BY s.snumber

      WHERE ANY (sp.snumber BY p.pnumber WHERE

            s.snumber = sp.snumber

            AND sp.pnumber = p.pnumber) = 0) = 0
```

• Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation parts.

```
RANGE OF p IS parts
APPEND TO p (pnumber = "P7", city = "ATHENS",
      weight = 24 )
```

• Q5. Delete supplier S1.

```
RANGE OF s IS supplier
      DELETE s WHERE snumber = "S1"
```

The fact that there are only four basic data manipulation operators in Quel means that it is a relatively easy to use. This is due to the fact that it is based on the relational data model which has a simple data structure. It is considered by Date (1987) to be technically superior to the language SQL which has been accepted by the American National Standards Institute (ANSI) as an official standard language for relational systems, because Quel is more closely based on the relational data model. Furthermore, it is easier to use and learn.

2.3.6 SQL

This was defined at the IBM Research Laboratory in California, where a prototype implementation was built for it. Since then, it has developed into an important and widespread language. SQL interfaces are being provided for a variety of systems, in addition to IBM products. Further, the American National Standards Database Committee has formally adopted a standard relational database language that is closely based on SQL.

The query language SQL includes both a data definition part and a data manipulation part. In the IBM relational DBMS, DB2, SQL is the query language and it is implemented at two different interfaces:

- an interactive interface

- an application programming interface.

This availability of the query language at more than one interface is becoming a significant feature in most relational systems. For example, in the Informix DBMS (Informix, 1986), the query language SQL is available as an interactive query language and also as an embedded query language ESQL. ESQL, consists of SQL statements embedded in the C programming language. In the DB2 implementation, SQL statements can be embedded in PL/I, Cobol, Fortran, and Assembler.

SQL is a set-level language, rather than a record-at-a-time language, and in this way is typical of a relational query language. There are four DML statements available in SQL. These are `SELECT`, `UPDATE`, `DELETE` and `INSERT`. Note that `SELECT` is the most fundamental of the four operations, since the others must be preceded by it either explicitly or implicitly. The set of example queries is now shown expressed in SQL.

- Q1. Get supplier numbers for suppliers who supply part P2.

```
SELECT  snumber
FROM shipments
WHERE pnumber = 'P2'
```

- Q2. Get supplier numbers for suppliers who supply at least one red part.

```
SELECT snumber
FROM shipments
WHERE pnumber  IN
      (SELECT pnumber
      FROM parts WHERE color = 'RED')
```

• Q3. Get supplier names for suppliers who supply all parts.

```
SELECT sname

FROM suppliers

WHERE

      (SELECT  pnumber

      FROM shipments

      WHERE snumber = supplier.snumber)

            =

      ( SELECT pnumber

      FROM parts)
```

• Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation `parts`.

```
INSERT

INTO parts (pnumber, pname, color, weight, city)

      VALUES ( 'P7', 'WASHER',  'GREY', 2, 'ATHENS' )
```

• Q5. Delete supplier S1.

```
DELETE

FROM suppliers

      WHERE snumber = 'S1'
```

There are only four DML statements in SQL. This is one of the reasons for the comparative ease of use of the language, and the fact that there are only four such operations is a consequence of the simplicity of the relational data structure. All data in a relational database is represented in exactly the same way, that is, as values in column positions within rows of tables. Since there is only one way to represent anything, only one operator is needed for each of the four basic functions, `RETRIEVE`, `CHANGE`, `INSERT` and `DELETE`. In addition, it is relationally complete.

The SQL format is keyword based. It has a sound mathematical basis and provides built-in functions. It is therefore a suitable target language. However, it is considered to be inferior to Quel since it omits certain relational features and lacks 'orthogonality', that is, it lacks consistency and there is an arbitrariness in syntax and query construction. Further, though it has been adopted as a standard, none of the existing implementations of SQL are either identical to each other or to the standard.

2.3.7 Query-by-Example (QBE)

This is another relational language, described in Zloof (1977), and it is intended as a graphical interface to databases. Each operation in QBE is specified using one or more tables. Each of these tables is built up on the screen with the user specifying some parts, and the system others. Since QBE specifies its operations in a tabular form, its syntax is described as being two-dimensional. The languages considered so far have linear syntax. QBE has been the target of usability testing and the results have been favourable (Shneiderman, 1980).

In QBE queries are expressed using examples. These are used to formulate queries by entering an example of a possible answer in the appropriate place in an empty table. QBE automatically eliminates redundant duplicates from query results. Some example queries in QBE are now given.

• Q1. Get supplier numbers for suppliers who supply part P2.

suppliers	snumber	sname	status	city
	P.sx			

shipments	snumber	pnumber	qty
	sx	P2	

In this query sx is used as a link between the relations suppliers and shipments. This is equivalent to a JOIN in the relational algebra, and an existential quantifier in relational calculus.

• Q2. Get supplier numbers for suppliers who supply at least one red part.

shipments	snumber	pnumber	qty
	P.sx	px	

parts	pnumber	pname	color	weight	city
	px		RED		

• Q3. Get supplier names for suppliers who supply all parts.

suppliers	snumber	sname	status	city
	sx	P.sn		

shipments	snumber	pnumber	qty
	sx	all.px	

parts	pnumber	pname	color	weight	city
	all.px				

The expression all.px in the table parts refers to the set of all part numbers present in this table, while the same expression in the shipments relation refers to the set of all part numbers supplied by the supplier sx. Since the two expressions are identical, sx must be the supplier who supplies all parts.

• Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation parts.

parts	pnumber	pname	color	weight	city
INSERT	P7	WASHER	GREY	2	ATHENS

• Q5. Delete supplier S1.

```
suppliers   snumber   sname   status   city

DELETE       S1
```

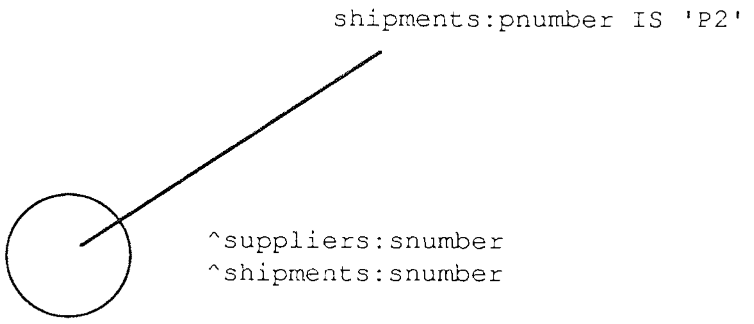
QBE has an essential feature which distinguishes it from other languages discussed in that it uses examples in the specification of queries. It also is different from many query languages in that it uses a simple two-dimensional syntax. This means that the user has the tables as a pre-established frame of reference. When originally specified, it was said not to lose its simplicity when used to express complex queries. However, later research does not agree with this view (see Cuff, 1982). QBE allows the user freedom to build up queries in any order that is suitable to the user. The rows and the order in which the user completes them in specifying the query is arbitrary, i.e. QBE is a highly nonprocedural language. It is based on relational calculus and is relationally complete. It is a relatively simple language to use, but has a number of problems due to its graphical representation. In the case of relations in the database with many attributes, the tables would not fit horizontally on the screen. Thus it is difficult to specify a query. Further, queries requiring the production of aggregate functions over a set of values which must be used for a further computation, cannot be stated satisfactorily. Since it is a graphical query language, it is ideally used as a visually oriented front-end to the database, conversely, for the same reason, it is not suitable as a target language. Although QBE has been implemented, it is not widely available.

2.3.8 GOING

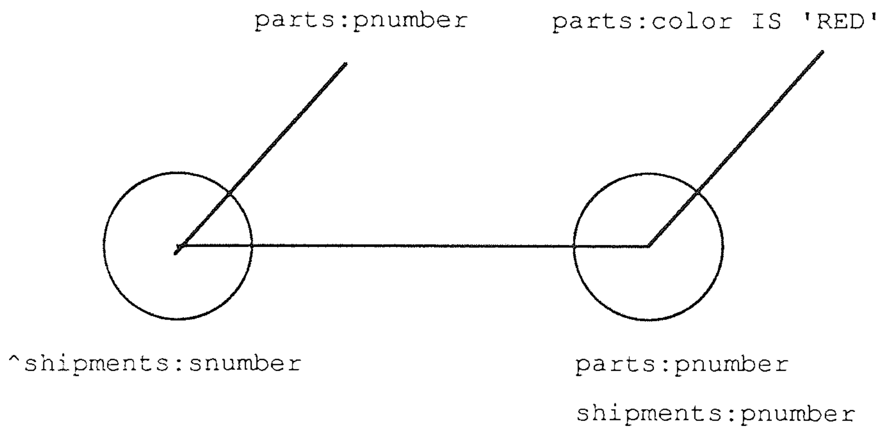
This is another of the graphical query languages for database systems (Udagawa & Ohsuga, 1982). It enables users to express queries in terms of nodes, arcs, comparison predicates and functions. Simple figures are used to express queries (ellipses represent domains, arcs for logical orders of entities and connection of conditions), along with expressions composed of comparison predicates and functions. It uses very simple English-like statements as a means of avoiding the difficult processing that would be involved for natural language, and avoids the use of quantifiers and bound variables in expressing queries. This eliminates the need for the user to have detailed knowledge of predicate logic. The user is provided with the means of controlling the size and layout of the graph.

In GOING queries, a domain or literal expression preceded by '^' specifies that its value is to be printed. Queries containing predicates are expressed in terms of domain specifications, boolean expressions and directed arcs.

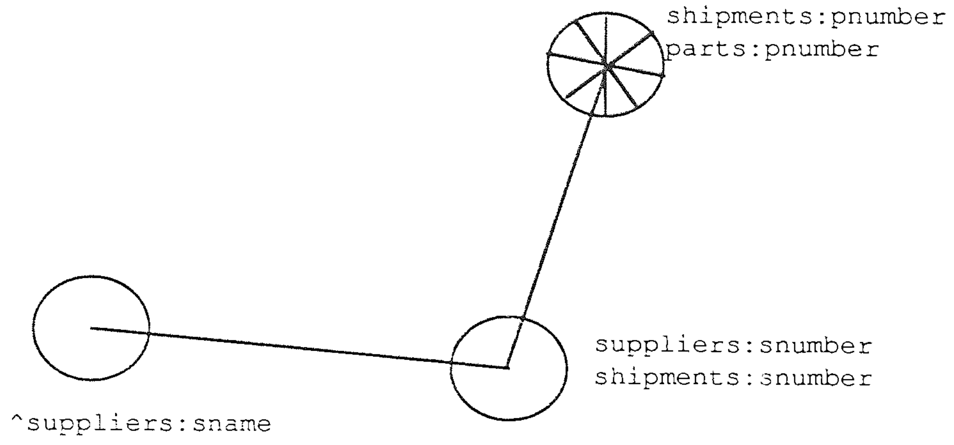
- Q1. Get supplier numbers for suppliers who supply part P2



- Q2. Get supplier numbers for suppliers who supply at least one red part



• Q3. Get supplier names who supply all parts



Q4. and Q5. are not describable, since the designers of GOING have not specified a syntax for updates. In the paper, discussion is limited to queries, however the underlying assumption is that updates are easily added to the language.

A feature of this language is that queries are stated in a nonprocedural way. The meaning of a query depends on the properties of its components and not the order in which it is made. Though it is designed to minimise the number of concepts that the user has to learn in order to use the whole language, it is a relatively complex language to use and is unlikely to be used by the class of users at which it is aimed that is, non-programmers. However, it is relationally complete. Its queries are translated into an intermediate language based on relational algebra. Because it is graphical, it is unsuitable as a target language. Another factor that could be detrimental to such use is the length of time taken to arrive at target queries. Compared to other graphical query languages, such as QBE and CUPID, its basic concepts, ellipses, arcs, nodes, and so on, are less familiar. Furthermore, no implementations of GOING are available.

2.3.9 CUPID

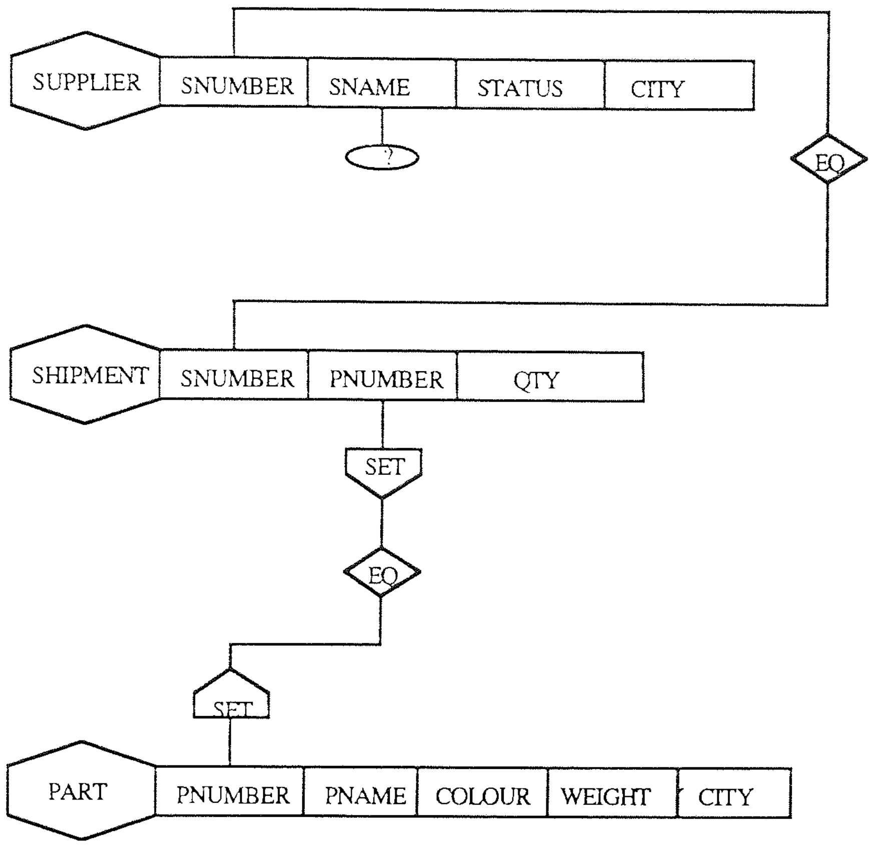
CUPID (Casual User Pictorial Interface Design) is also a graphical database interface (McDonald & Stonebraker, 1975) and was specifically developed for a particular class of user, being designed as an experimental system for casual and possibly infrequent use by non-programmers. This is implemented as a front-end to Quel, which is the query language for the relational DBMS Ingres which runs under Unix.

The user can construct queries by light-pen manipulation of a number of standard symbols. The graphical representation is of the query rather than of the database (as in Foral LP). It translates the diagram that the user creates into a formal linear language Quel, which is then passed to the underlying relational DBMS. The screen is divided into three areas which are:

- instructions on how to proceed,
- a menu of the commands, and
- a working area.

Queries are built up by combining table selection operations and query drawing operations. Because of the complexity in expressing CUPID queries, only one example query is given. However, the other queries are represented in a similar manner.

• Q3. Get supplier names for suppliers who supply all parts.



The main problem with queries expressed in this language is the effort needed to specify the queries. A relatively simple query can take about fifty light-pen hits to specify. It is meant for users with no programming knowledge, yet complex queries still use logical predicates and involve navigational knowledge. An obvious advantage is that queries that have been expressed can be seen pictorially. But the more technical user may find it cumbersome and impractical for specifying queries. However, it is relationally complete and has a sound basis in that its queries translate to the relational calculus based query language Quel. It is not a suitable target language. This is because it is a highly visual language and any choices made during its design would be to enhance and support this aspect. These are not the choices that would make it suitable as an internal intermediate language. This query language is known to have been implemented by researchers, but it is not widely available and is further limited by its need of specialised equipment (light pen and bit-mapped screen).

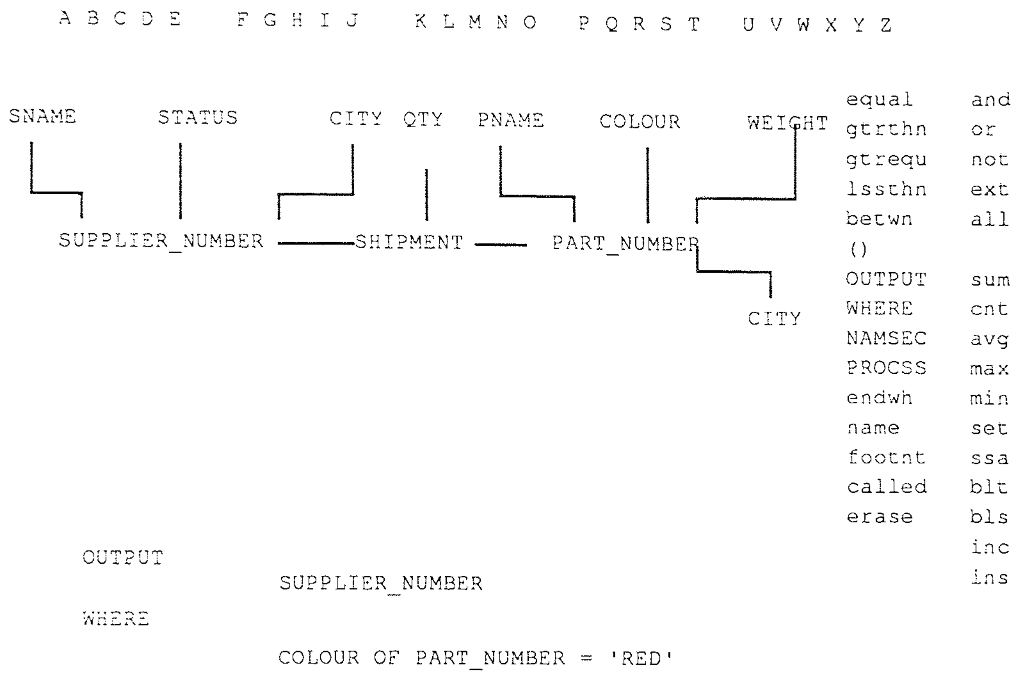
2.3.10 Foral LP

Foral LP (Foral with a Light Pen) was designed as an experimental system combining menu selection with a displayed network representing a database. The users main communication device is a light pen, which enables the user to specify:

- database attributes,
- linkage paths between them, and
- procedural and logical operations on them.

Constants are also specified using the light-pen, and for this purpose the alphabet and digits are permanently displayed across the top of the screen. A constant can be specified by selecting repetitively from this set, the idea being that the user is not expected to have any typing ability or access to a keyboard. The major part of the display area shows a network representation of the database. There is also a list of mnemonics for logical comparatives and connectives, built-in functions, arithmetic functions and instructions for the Foral LP interpreter. Finally, an area is reserved at the bottom of the screen, in which will appear a linear language representation of the developing query. As for Cupid, only one example of a Foral LP query is given.

• Q2 Get supplier numbers for suppliers who supply at least one red part



```
OUTPUT
            SUPPLIER_NUMBER
WHERE
            COLOUR OF PART_NUMBER = 'RED'
```

As with QBE and CUPID, this language has the advantages of visual representation of the query and the database, and is targeted at a particular class of user. This is an imaginative approach to query languages and it has a distinct advantage in that it frees the user to a certain extent from having to know how the database is split into relations; the network representation on the screen showing logical connections between entities. This means that the physical implementation may be changed without affecting the user, and it enables the user to visualise the data in the database in terms of the real-world entities and relationships, rather than in the rather limiting terms of their computer representations.

Despite these advantages, it is still better suited to a trained, regular user, since practice and skill are required for anything other than very simple queries. Again, there is a problem of graphical representation in that a simple logical structure shows up clearly, but a more complicated one looks confusing and is limited by screen size. This aggravates the problem of the level of accuracy required for making hits with the light-pen. The interface itself is meant as a high-level front-end to a DBMS, and therefore does not lend itself to supporting a front-end. Again, specialised equipment requirements limit its availability.

2.3.11 FQL

FQL embodies many of the ideas concerning functional programming systems described by Backus (Backus, 1978). The only control structure available to the user is the ability to combine functions. FQL differs from other query languages in several important respects. These are:

- there is no notion of data currency (note however that this is also true of the relational query languages).
- complex queries may be developed incrementally from simpler queries: a query in FQL is no more than another function over the database.
- full computational power is provided: (many query languages lack the ability to do basic arithmetic).

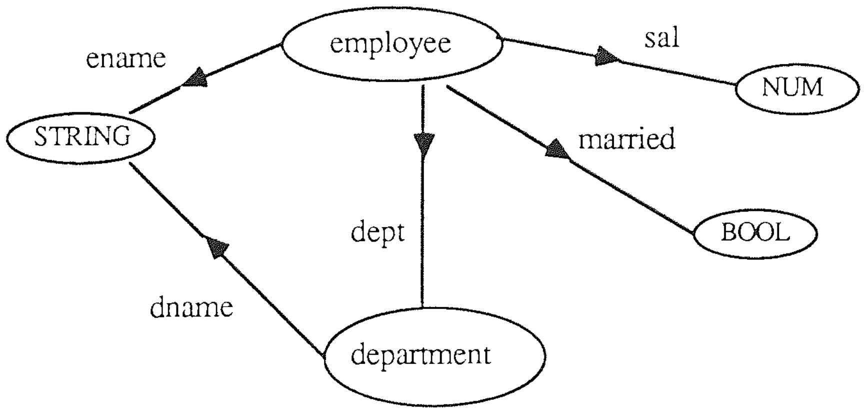


Figure 2.1: An Example FQL Schema

Figure 2.2 shows a schema for a very simple database. The database is regarded as a collection of functions over various data-types. The example schema consists of two entities, these are of type `employee` and of type `department`. The function `dept` for example, represents a mapping between these entity types. That is, given an `employee`, the function `dept` returns the `department` in which he works. The remaining functions map into basic types. For example the function `ename` returns a `STRING` which is the name of a particular employee entity. This schema can also be expressed as:

```
dept        : employee      -> department
ename       : employee      -> STRING
sal         : employee      -> NUM
married     : employee      -> BOOL
dname       : department    -> STRING
```

The data types `STRING`, `BOOL` and `NUM` are standard, that is, they exist independently of any database, while the types `employee` and `department` and the five functions described are specific to this database. Information about `employees` and `departments` may only be obtained through those database functions which map these entities into 'printable' types, such as `STRING`.

On the basis that the database defines a set of functions, the language provides mechanisms for combining functions to create new and more powerful functions. The basic mechanism is composition. This defines a new function in terms of other already existing functions. For example, a function could be defined which given an `employee` entity returns the department in which he works:

```
deptname    : employee -> STRING = dept.dname
```

This FQL function definition defines `deptname` to be a function from `employee` to `STRING`, and it is defined to be the composition of the functions `dept` and `dname`.

Composition is denoted by the use of the fullstop. The functions are evaluated left to right. This means that the function dept is applied to an employee entity, producing a department entity. This department entity has applied to it the function dname to produce a value of type STRING.

The functions considered so far do not return collections of objects. Inverse functions provide the means to do this. The inverse of dept, which would be written as !dept, is a function which maps a department into a sequence of all those employees who belong to a given department. The FQL terminology for such a sequence is a stream. New functions can be created which map streams into other streams. The functional forms provided to do this are extension and restriction. Extension allows a function such as sal which maps an employee to a NUM to be extended into a function *sal which, given a stream of employees, returns a stream of NUMS by applying the function sal to each employee within the stream. Restriction allows streams to be filtered by predicates over individual elements. For example, the function |married maps a stream of employees into a substream of employees satisfying the condition that they be married. Extension will preserve the length of a given stream, while restriction will generally return fewer elements. Consider the the creation of a function which returns a stream of salaries of all married employees within a given department. This would be achieved as follows:

```
married_sals: department ->
        *num = !dept.|married.*sal
```

The remaining functional form is construction. This is used to create functions that return tuples of objects, for example, an employee's name and salary. This would be expressed as:

```
name_and_sals: employee -> [STRING, NUM] = [ename,
        sal]
```

The notation `[ename, sal]` specifies a mapping from `employee` to a pair comprising of STRING and NUM.

In FQL, a query is a special kind of function whose range is some printable object. For example, to find the department names and salaries of all married employees, access is needed to the stream of all employees within the database. To enable this, the functional view of the database needs to be extended to include a set of 'constant' functions. In the database example these would be:

```
!employee          : -> *employee

!department        : -> *department
```

As can be seen by the above specification, a constant function is denoted by the absence of a data type to the left of the arrow symbol. The query would then be expressed as :

```
Q: ->*[char, num] = !employee.|married.*[dept.dname,
      sal]
```

The class of queries that can be formulated using only the functions given by the database is limited. The language therefore contains standard functions, which include arithmetic, relational and boolean operators. Some example queries using the FQL language are given. The suppliers-and-parts database that we have used so far to illustrate the various query languages can be expressed using FQL notation as follows:

```
snumber     :suppliers -> CHAR

sname       :suppliers -> CHAR

status      :suppliers -> INT

city        :suppliers -> CHAR

pnumber     :parts -> CHAR

pname       :parts -> CHAR
```

```
color        :parts -> CHAR

weight       :parts -> INT

city         :parts -> CHAR

sshipments   :pnumber -> snumber

pshipments   :snumber -> pnumber

qty          :[snumber, pnumber] -> INT

!suppliers   : -> *suppliers

!parts       : -> *parts
```

• Q1. Get supplier numbers for suppliers who supply part P2.

```
!suppliers. |  ([ pnumber, "P2']  . EQ) . *snumber
```

• Q2. Get supplier numbers for suppliers who supply at least one red part.

```
!parts . | ( [color, "RED"] . EQ) . pnumber . |

      sshipments . *snumber
```

• Q3. Get supplier names for suppliers who supply all parts.

For this query another definition for the database is required. That is:

```
samesupp    : [snumber, pnumber] -> ( [snumber,

      snumber] . EQ)
```

The query can then be expressed:

```
( [!parts . pnumber, !samesupp . pnumber] . EQ ) |

      !sshipments . *sname
```

The queries Q4 and Q5 are not shown, since the definition of FQL does not contain the syntax for updating the database.

FQL is a precise and powerful formalism for the expression of database queries, though it is not an ideal end-user language. Because of its power and precision it is probably best suited to being used as an intermediate language or target language into which other query languages are translated. Further, it is limiting as a general database applications design language because it does not have the ability to update functions. It also lacks any mechanism for specifying and querying meta-data. It has no clearly defined means for specifying constraints, though it does have the means to specify derived data. If relational completeness is applied to it loosely, it can be considered to be 'relationally complete', in that it is selectively powerful. Its availability is also limited. It is at a lower level of abstraction than, for example, Daplex.

2.3.12 Daplex

This is the query language defined by Shipman (1981) for his functional data model. The term is used by Shipman to describe both the query language and the data model described in the paper. The language will be described briefly here and will be discussed in more detail in Chapter 3.

The language Daplex has a simpler notation than FQL, the other functional model based query language discussed. A variation of Daplex has been implemented in an experimental system (Kulkarni, 1983). A subset of Daplex has been embedded in Ada. This system is called Adaplex and is described in Smith *et al.* (1981b). Another project (Smith *et al.*, 1981a) is underway using Adaplex as the common language to access heterogeneous distributed databases.

Essentially Daplex supports query formulation based on the set and function operators. In the Daplex functional data model, data is modelled in terms of entities. Database entities bear a one-to-one correspondence to real world entities. Relationships between data are expressed as functions. There are three classes of functions on entities as defined by Shipman (1981). These are as follows:

- scalar-valued - unlike entities, these are directly printable, and correspond to the usual basic types used in programming languages.
- single-valued - these functions return a single entity.
- multi-valued - these functions return a set of instances of a particular entity.

Daplex incorporates both data definition facilities and also data manipulation facilities. A Daplex representation of the sample suppliers-and-parts database would be as follows.

```
DECLARE suppliers( ) ->> ENTITY
DECLARE snumber(suppliers) -> STRING
DECLARE sname(suppliers) -> STRING
DECLARE status(suppliers) -> INTEGER
DECLARE city(suppliers) -> STRING

DECLARE parts( ) ->> ENTITY
DECLARE pnumber(parts) -> INTEGER
DECLARE pname(parts) -> STRING
DECLARE color(parts) -> STRING
DECLARE weight(parts) -> INTEGER
DECLARE city(parts)-> STRING

DECLARE shipments( ) ->> ENTITY
DECLARE snumber(shipments) -> STRING
DECLARE pnumber(shipments) -> STRING
DECLARE qty(shipments) -> INTEGER
```

Having described the relational schema, it is necessary to define the functional view:

```
DEFINE supplier(parts) -> suppliers SUCH THAT
      FOR SOME shipments
            snumber(suppliers) = snumber(shipments) AND
                  pnumber(shipments) = pnumber(parts)

DEFINE part(suppliers) -> parts SUCH THAT
      FOR SOME shipments
            pnumber(parts) = pnumber(shipments) AND
                  snumber(shipments)
                  = snumber(suppliers)
```

- Q1. Get supplier numbers for suppliers who supply part P2.

```
FOR EACH suppliers SUCH THAT
      pnumber(parts) = 'P2'
PRINT snumber(suppliers)
```

- Q2. Get supplier numbers for suppliers who supply at least one red part.

```
FOR EACH suppliers SUCH THAT
      color(part(suppliers)) = "RED"
PRINT snumber(suppliers)
```

- Q3. Get supplier names for suppliers who supply all parts.

```
FOR EACH suppliers SUCH THAT
            part(suppliers) = supplier(parts)
PRINT sname(suppliers)
```

- Q4. Add part P7 (name 'WASHER', color 'GREY', weight 2, city 'ATHENS') to relation `parts`.

```
FOR A NEW parts
      BEGIN
            LET pname(parts) = "P7"
            LET color(parts) = "RED"
            LET weight(parts) = 2
            LET city(parts) = "ATHENS"
      END
```

• Q5. Delete supplier S1.

```
FOR THE suppliers SUCH THAT
      sname(suppliers) = "S1"
      BEGIN
            EXCLUDE snumber(suppliers)
            EXCLUDE sname(suppliers)
            EXCLUDE status(suppliers)
            EXCLUDE city(suppliers)
      END
```

The language Daplex provides facilities for data definition as well as data update. Its data model provides for a complete set of operations. In addition, it has facilities for specifying constraints and also derived data. Daplex can be used to express data models other than the functional data model (Gray 1984, Stocker *et al.* 1984). Its format is keyword based and relatively easy to learn and use. Although it is not a relational query language, its selective power is high. Due to its well-defined data model it could be used successfully as a target language, but this is limited by its lack of availability.

2.4 CONCLUSIONS

The relational model is suitable as a target model since it has a sound basis. It is a data model on which most DBMS are based and is relatively easy to use. It is a very useful conceptual tool, which is a proven basis for high level set-oriented query languages for querying and updating the database. DBMS based on the relational model are characterised by the fact that they enhance the ease of use and data independence aspects of databases. This leads to a reduction in the cost of database intensive application programming. Relational model-based DBMS also provide a good environment for back-end support and provide interfaces that are very robust (Tsur & Zaniolo, 1984 and Zaniolo, 1983).

However, a limitation of the relational model is that schemas in it fail to model completely and expressively the natural relationships and mutual constraints between entities. As a result, other data models have been proposed where reality is modelled in terms of entities and relationships among entities. Some of these have been discussed in this chapter.

If the approach were taken to completely abandon relational DBMS and build new systems based on the the new model, little support would be found, since it would be very expensive in terms of time, cost and manpower. There is considerable investment in terms of all three in current database technology and it would be unreasonable to expect this approach to be taken.

It would be more useful to provide other data models as front-ends to existing relational databases and add the capabilities of the new approach while still retaining the old ones. This approach has the advantage that compatibility is ensured, existing skills and knowledge are preserved and a choice is provided to users.

The relational DBMS Ingres has been chosen for the basic DBMS, since it is amongst the most widely used relational DBMS, and its query language, Quel, makes an appropriate target language. Quel is relationally complete and is based on relational calculus. It is relatively nonprocedural and provides a core of features that will be required in some form or another in all such languages. Relational languages are formally specifiable and are therefore a good basis for an interface.

The functional data model and query language Daplex provide a high-level interface to other data models and query languages as evidenced by Katz & Goodman (1983). A high level interface based on Daplex means that the definition and manipulation of complex objects is made simpler, since Daplex has only two basic constructs: entities and functions. Entities represent real world objects and functions map entities to entities. In addition, Daplex provides the use of function composition for the traversal of complex objects.

Functional data models in general provide a 'simple' (note that this is not necessarily the same as 'user friendly') data manipulation language (Kulkarni, 1983) and are considered to provide a natural mode of expressing queries. As with the relational model, the functional data model has a sound mathematical basis (the mathematical theory of functions).

Daplex is a precise and powerful mechanism for expressing database queries. Since Daplex has capabilities which incorporate those of the other data models widely used today, it is viable as a front-end to databases based on these other data models (Shipman, 1981). Compared to other extended data models, Daplex has retained some of the advantages of the relational model, in that it is relatively easy to use and implement.

By providing these two query language interfaces to the same database means that users can regard the database as a collection of relations or conversely as a collection of functions. Queries in Daplex can be mapped into equivalent Quel expressions which in turn map into relational calculus. This method of mapping Daplex into Quel queries is convenient and provides a basis for portability.

CHAPTER 3

THE IMPLEMENTATION ENVIRONMENT

3.1 INTRODUCTION

This chapter consists of a discussion of the implementation environment for the front-end system, and the reasons for selecting this environment. The query language Daplex which is based on the functional data model defined by Shipman (1981) is the query language available in the front-end. This is interfaced to the relational DBMS Ingres, which has as its query language Quel. The operating system upon which the system is implemented is the Unix system (Kernighan & Pike, 1984). The implementation is on an BT M6000 series supermicro running Unix System V.2.

3.2 INGRES

3.2.1 Background

Ingres (Interactive Graphics and Retrieval System) is a relational database system which is implemented on the Unix operating system. It is described in, for example, Stonebraker *et al.* (1976 and 1982) and Stonebraker (1984). It was developed at the University of California at Berkeley. The Ingres prototype became widely available in university environments in the early years of its development (late 1970s and early 1980s). This version is now usually called 'University Ingres', to distinguish it from the commercial version that has since become available. University Ingres is still the basis of active research and development at Berkeley. However in the early 1980s, a company called Relational Technology Inc. (RTI) was formed to develop and market a commercial version of Ingres. Both versions of Ingres run under Unix. The two versions of Ingres are similar in most respects.

Ingres is primarily programmed in C, which is the high level language in which Unix itself is written. Its designers were motivated to use the relational model of data due to the following reasons:-

- the high degree of data independence that such a model affords,
- the possibility of providing high level, procedure free facilities for data definition and access.

The primary user language in Ingres is Quel which has been briefly introduced in Chapter 2, section 3.4 and is described in more detail in this chapter. Ingres also has the language Quel available embedded in other high-level programming languages (as Equel). This is to provide the flexibility of a general purpose programming language in addition to the interactive facilities provided by Quel. All Quel statements are valid Equel statements, therefore Equel will not be discussed in detail.

3.2.2 Invoking Ingres

Ingres can be invoked in two ways: either by directly executing Ingres and supplying a database name, that is:

```
ingres <dbname>
```

or by executing an applications program which has Equel statements embedded in it.

3.2.3 Structure

When the Ingres DBMS is executing, the process structure shown in figure 3.1 is created.

Process 1 is an interactive terminal monitor which allows a user to specify, print, edit, and execute collections of Ingres commands. It does this by maintaining a workspace for the user, which the user interacts with until he is satisfied with the interaction. The contents of the workspace are passed down the pipe A as a string of ASCII characters when execution is required. Processes 2, 3 and 4 consist of various query processing routines.

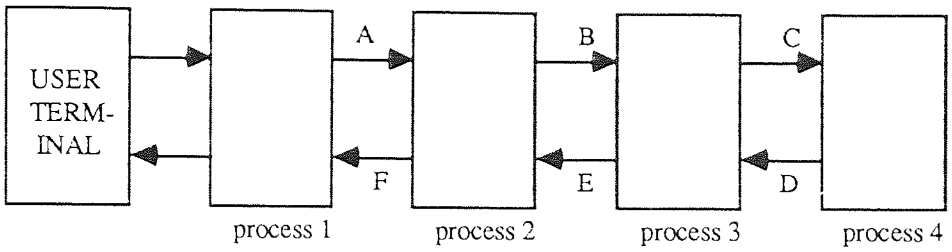


Figure 3.1: Ingres Process Structure

3.2.4 Quel

Quel will now be considered in more detail.

3.2.4.1 Data Definition

Quel has the following data definition statements:

• `CREATE`	- to create a table
• `INDEX`	- to create an index
• `DEFINE VIEW`	- to create a view
• `DESTROY`	- to delete a table, index or view
• `MODIFY`	- to change the storage structure of a base table or index.

3.2.4.2 Retrieval Operations

These are:-

• `RETRIEVE`

- REPLACE
- DELETE
- APPEND

Quel interactions include at least one RANGE statement. The form of this is:

```
RANGE OF variable-list IS relation-name
```

This statement specifies the relation over which each variable ranges. The `variable-list` in a RANGE statement declares variables which will be used as arguments for tuples. These are known as tuple variables. Operations will also include one or more statements of the form:

```
COMMAND [result-name] (target-list) [WHERE
      qualification]
```

where COMMAND is one of the four operations specified above. For RETRIEVE and APPEND operations, `result-name` is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE, `result-name` identifies the name of the tuple variable which is to be amended if it meets the qualification. For DELETE, `result-name` identifies the tuple variable that is to be deleted if it satisfies the qualification. The `target-list` has the following form:

```
result-domain = QUEL function.......
```

The examples in this discussion of Quel will be based on the `suppliers-parts` database described in chapter 2. A qualified retrieval in this language, for example:

• Get details of suppliers in Paris with a status of more than 20, would be expressed as follows:

```
RANGE OF s IS suppliers
RETRIEVE s.snumber, s.status
WHERE s.city = "PARIS"
AND s.status > 20
```

In this query `s` is the tuple variable which ranges over the `suppliers` relation. The results of the query are displayed on the screen, however the results can be saved in a table if required. The above query amended to store the result in a table called `newtable` would then be expressed as:

```
RANGE OF s IS suppliers
RETRIEVE INTO newtable s.snumber, s.status
WHERE s.city = "PARIS"
      AND s.status > 20
```

Queries can be made which involve more than one table (that is a join of tables). A simple example of this is:

• Find all the details for all suppliers and parts that are located in the same city. This would be expressed as:

```
RANGE OF s IS suppliers
RANGE OF p IS parts
RETRIEVE s.all p.all
WHERE s.city = p.city
```

The above query shows the join of two different tables, however queries can be expressed that involve a table joining with itself. An example query would be:

- Retrieve all pairs of supplier numbers such that the two suppliers concerned are located in the same city.

```
RANGE OF s1 IS suppliers
RANGE OF s2 IS suppliers
RETRIEVE s1.snumber, s2.snumber
WHERE s1.city = s2.city
      AND s1.snumber ≠ s2.snumber
```

In this query the two range variables `s1` and `s2` have been specified to range over the same table, `suppliers`.

3.2.4.3 Aggregation

The language Quel provides a number of aggregate operators such as `COUNT`, `SUM`, `MAX`, `MIN`, and so on. These operate on the collection of values in a particular column of a particular table. The general syntax for an aggregate reference is:

```
AGGREGATE ( expression [WHERE predicate ] )
```

An aggregate operator returns a single scalar value and can therefore appear in the target list or in the `WHERE` clause wherever a constant is allowed. The following three queries make use of aggregates.

- Get supplier numbers for suppliers with a status value less than the current maximum status value in the `suppliers` table.

```
RANGE OF s IS suppliers
RETRIEVE s.snumber
WHERE s.status < MAX (s.status)
```

The range variable appearing inside the argument to an aggregate is local to that aggregate and is therefore distinct from any range variables which appear outside the aggregate. This is true even if the range variables have the same name, as is the case with the above example. The query could also have been expressed as:

```
RANGE OF s1 IS suppliers

RANGE OF s2 IS suppliers

RETRIEVE s1.snumber

WHERE s1.status < MAX (s2.status)
```

3.2.4.4 Aggregate Functions

An aggregate function is an aggregate operator which has an argument that includes a BY clause. It is distinguishable from an aggregate since its value is a set rather than a single scalar. An example which uses an aggregate function is the query:

- Get the part number and the total shipment quantity for each part supplied.

```
RANGE OF sp IS shipments

RETRIEVE sp.pnumber, X = SUM (sp.qty

      BY sp.pnumber)
```

Queries can be expressed that have an aggregate function in the WHERE clause, for example:

- Get part numbers for all parts supplied by more than one supplier:

```
RANGE OF sp IS shipments

RETRIEVE sp.pnumber

WHERE COUNT( sp.snumber BY sp.pnumber) > 1
```

3.2.4.5 Quantification

A query which uses existential quantification, such as:

• Get supplier names such that there exists a shipment record with the same supplier number and with part number P2, would be expressed:

```
RANGE OF s IS suppliers
RANGE OF sp IS shipments
RETRIEVE s.sname
WHERE s.snumber = sp.snumber
AND sp.pnumber = "P2"
```

This quantification can be expressed more explicitly using the aggregate function ANY as follows:

```
RANGE OF s IS suppliers
RANGE OF sp IS shipments
RETRIEVE s.sname
WHERE ANY (sp.snumber BY s.snumber
      WHERE s.snumber = sp.snumber
            AND sp.snumber = "P2") = 1
```

The ANY aggregate function returns the value zero if its argument set is empty, otherwise it returns the value one. As can be seen from the above query the use of the ANY aggregate makes the expressed query more complex. However the ANY aggregate is most useful in the form that checks to see if the returned value is zero, that is the negated form of the existential quantifier. As an example consider the query:

• Get the names of suppliers who do not supply part P2.

```
RANGE OF s IS suppliers
RANGE OF sp IS shipments
      RETRIEVE s.sname
            WHERE ANY (sp.snumber BY s.snumber
                  WHERE s.snumber = sp.snumber
                        AND  sp.pnumber = "P2") = 0
```

3.2.4.6 Updates

The Quel language provides three update operations which are REPLACE, DELETE, and APPEND. The syntax for these follows the general syntax given above. These will be illustrated by means of examples.

• REPLACE - for example, change the colour of part P1 to red and decrease its weight by a factor of two:

```
RANGE OF p IS parts
REPLACE p ( color = "RED", weight = p.weight - 2)
      WHERE p.pnumber = "P1"
```

• Double the status rating of all suppliers in London:

```
RANGE OF s IS suppliers
REPLACE s (status = 2 * s.status)
WHERE s.city = "LONDON"
```

• APPEND - for example, add a part P8 to the table PART. The city is Athens, weight is 24 and the other attributes are not yet known:

```
RANGE OF p IS part

APPEND TO p ( pnumber="P8", city="ATHENS",

      weight=24)
```

• DELETE - for example, delete all suppliers in Athens:

```
RANGE OF s IS suppliers

DELETE s WHERE s.city = "ATHENS"
```

3.2.4.7 Views

Quel provides for view definition, the general syntax is as follows:

```
DEFINE VIEW viewname (target-list) [ WHERE predicate]
```

An example view definition for the database would be to provide a relation which consisted of all those suppliers based in Paris. This would be expressed in Quel as:

```
RANGE OF s IS suppliers

DEFINE VIEW PARIS_SUPPS

      (snumber = s.snumber,

      sname = s.sname,

      status = s.status,

      city = s.city)

WHERE s.city = "PARIS"
```

After such a definition has been made it is possible to make queries against this as if it was a base relation, for example:

• Retrieve all suppliers from this view with a status rating less than 25 would be expressed:

```
RANGE OF ps IS paris_supps

RETRIEVE ps.ALL

WHERE ps.status < 25
```

The query will be modified internally to be a query over the original base relation that was used in the view definition.

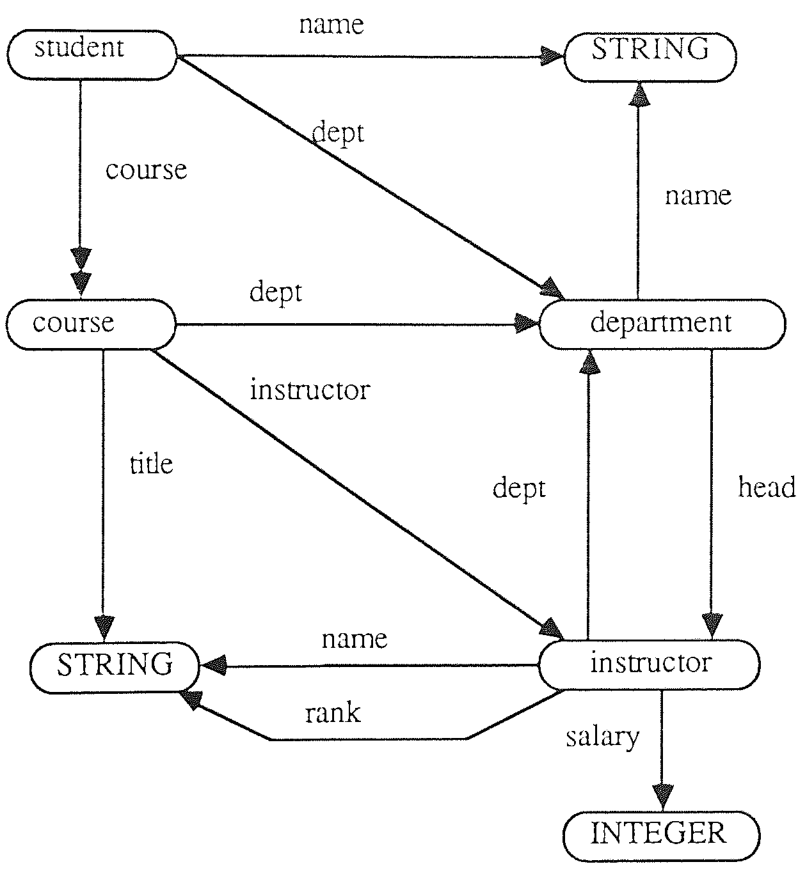


Figure 3.2: Daplex Data Model - The University Database

3.3 DAPLEX

3.3.1 Background

In this section the Daplex data model (introduced in section 2.2.1), and the Daplex query language (introduced in section 2.3.12) are discussed in more detail. In Daplex, data is modelled in terms of entities, where entities bear a one-to-one correspondence to the real world entities. Relationships between entities are expressed as functions. An entity is some form of token identifying a unique object in the database and usually represents a unique object in the real world. Figure 3.2 is an illustration of a Daplex data model, taken from Shipman (1981). Figure 3.3 shows the corresponding data description in Daplex for the database modelled in figure 3.2.

3.3.2 Entities

In the Daplex data model a student in the real world is represented by a unique `student` entity in the database. Entities with some common characteristics are classified as entity types. These entity types are in turn part of a type hierarchy, so that they are all subtypes of the type entity. Functions map a given entity into a set of target entities. Consider the function `student` shown in figure 3.3. This function evaluates to a set of entities of type entity. Since this function has no arguments there is only one possible result set. In this model this denotes the fact that members of this set have a distinct type. In this statement, the name student is 'overloaded', since it does all of the following:

- it names the entity type,
- it names the set of person entities,
- it names the function that produces that set.

```
DECLARE student( ) ->> ENTITY

DECLARE name(student) -> STRING

DECLARE dept(student) -> department

DECLARE course(student) ->> course


DECLARE course( ) ->> ENTITY

DECLARE title(course) -> STRING

DECLARE dept(course) ->department

DECLARE INSTRUCTOR(COURSE) -> INSTRUCTOR


DECLARE instructor( ) ->> ENTITY

DECLARE name(instructor) -> STRING

DECLARE rank(instructor)  -> STRING

DECLARE dept(instructor) -> department

DECLARE salary(instructor) -> INTEGER


DECLARE department( ) ->> ENTITY

DECLARE name(department) -> STRING

DECLARE head(department) -> instructor
```

Figure 3.3: Data Description

3.3.3 Functions

As introduced previously, there are three classes of functions. These are as follows:

- scalar-valued functions, which are directly printable unlike entities. They correspond to the usual basic types used in programming languages. For example, the function `name`, when given a student entity as an argument, returns a string which represents that student entity's name.
- single-valued functions, which return a single entity. For instance, the function `head` when given a `department` entity as an argument returns a unique entity representing the head of a university department.
- multi-valued functions, which return a set of instances of a particular entity. These are distinguished on the diagram by the use of the double-headed arrows. For example, consider the function `course` when given a `student` entity as an argument. This returns a set of courses (zero or more) which are being taken by that particular student.

The model also allows multi-argument functions and these provide a convenient means to establish relationships involving more than two entities without introducing artificial entities.

3.3.4 Data Definition

Shipman's functional data model has Daplex as its integrated data definition and data manipulation language. Shipman (1981) does not specify any general purpose computation facilities for Daplex, it is envisaged that Daplex would be embedded in a high level language and that this would provide the necessary facilities.

New functions can easily be added to the schema at any time. The DECLARE statement is used to add a base function or a base entity type. This statement establishes functions in the system. Functions are used to specify both entity types and also to express the properties of an entity. For instance, in the set of statements in figure 3.3 the statement:

```
DECLARE name(student) -> STRING
```

expresses the following information. It states that `name` is a function which returns entities of type `STRING` when given an argument entity of type `student`. This function happens to return a scalar entity, however functions can return non-scalar entities as well. An example of this is the following statement from figure 3.3:

```
DECLARE dept(student) -> department
```

This declares the function dept which returns an entity of type `department` when applied to a `student` entity. The distinction between this and the previous declaration is that in the second case a `department` entity is returned, not a department number or other attribute that is printable (as in the first case).

The above two declarations were for single-valued functions since they always return a single entity. These are indicated by the use of a single headed arrow. Multi-valued functions are declared as follows:

```
DECLARE course(student) ->> course
```

The use of the double-headed arrow (both in the above statement and in figure 3.2) indicates the function being declared is multi-valued, that is the `COURSE` function returns a set of entities of type `course` when given a `student` entity as an argument. In Daplex all function applications evaluate to sets of entities in the mathematical sense. This means that sets are considered unordered and do not contain duplicates.

As can be seen from figure 3.3, functions can be declared that do not take any arguments. Consider, for example, the statement:

```
DECLARE course( ) ->> ENTITY
```

This states that the function `course` evaluates to a set of entities. The convention in Daplex is that zero-argument functions define basic entity types. The example declaration statement is therefore doing the following:

- it is declaring a function called `course`

- it is defining the entity type `course`

Daplex also provides for the creation of multiple-argument functions. The data description in figure 3.3 could have included the following declaration:

```
DECLARE grade(student, course) -> INTEGER
```

This would declare the function `grade` to return the grade which was obtained by the `student` in a particular `course`. Other data models often force the creation of new entity types to express such a situation. An example of this is the entity-relationship model. In this situation it would be necessary to regard the enrollment of a student on a course as a conceptual object, and then to assign a `grade` property to that object. This could also be done in Daplex, but is avoidable. There is however a problem with the above declaration, in that it specifies the function `grade` as well defined for every `student-course` pair, whereas it only exists for those courses in which the student is enrolled. A declaration can be made which overcomes this problem, that is

```
DECLARE grade(student, course(student)) ->> INTEGER
```

This declares the function `grade` to exist for only those courses for which the student is enrolled. In the Daplex data model function name overloading is allowed. This means that more than one function may have the same name. An example of this name overloading are the functions `course()`, `course(student)`, and `course(staff)`. These all have the same name but in this model they are distinguished by their internal names. The internal name is obtained by enclosing the external function and the argument types over which it was originally specified, in square brackets. The above three functions therefore have the following distinguishable internal names: `[course()]`, `[course(student)]`, and `[course(staff)]`.

3.3.5 Derived Functions

The functions that have been discussed so far are introduced by a `DECLARE` statement and are called base functions. In the functional model base functions are represented by physically storing a table of arguments and results. Such functions are evaluated by executing an algorithm to search the list of arguments to determine the corresponding result value.

Functions introduced by the `DEFINE` statement are called derived functions. These are represented by an algorithm to compute their results. This means that the data for these functions does not exist explicitly in the database and is evaluated by executing the corresponding algorithm when required.

3.3.6 Type Hierarchy

The data description of figure 3.3 can be supplemented by the declarations given in figure 3.4. The declarations for `student` and `instructor` entities can be replaced by more general specifications. Thus, `student` and `employee` entities can be defined as `person` entities, while `instructor` entities are defined as `employee` entities. This implies a type hierarchy. For instance the entity type `student` is a subtype of the entity type `person`. These subtype-supertype relationships can be extended to any level and a particular entity type can have any number of subtypes. There are two important consequences of this hierarchical relationship structure, which are:

- an instance of an entity type is also an instance of its supertypes
- a subtype inherits all the functions defined over all its supertypes.

```
DECLARE person( ) ->> ENTITY

DECLARE name(person) -> STRING


DECLARE student( ) ->> person

DECLARE dept(student) -> department

DECLARE course(student) ->> course


DECLARE employee( ) ->> person

DECLARE salary(employee) -> INTEGER

DECLARE manager(employee) ->> employee


DECLARE instructor( ) ->> employee

DECLARE rank(instructor) ->STRING

DECLARE dept(instructor)->> department
```

Figure 3.4: Subtypes, Supertypes Declarations

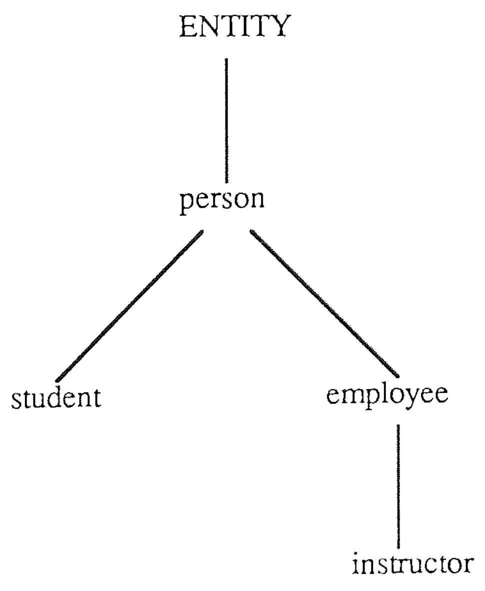


Figure 3.5: Type Hierarchy

3.3.7 Inverse Functions

The Daplex data model allows for mapping functions in two directions. For instance, the function `instructor` may be applied to a `course` entity to determine the `instructor` who teaches that particular `course`. However the model also allows for defining how to obtain the `course` taught by a particular `instructor` entity. This would be done by defining the following derived function:

```
DEFINE course(instructor) ->>
        INVERSE OF instructor(course)
```

This declaration means that a function now exists which maps `course` entities to `instructor` entities.

3.3.8 Order

Multi-valued functions evaluate to sets of entities; that is the sets are considered to be unordered and do not contain duplicates. However the concept of order may be natural for particular users, therefore Daplex provides the means for explicitly associating an order with entity types or multi-valued functions. This is done by using the order construct, for example:

```
DECLARE student( ) -> person IN ORDER BY ASCENDING
        name(student)
```

Further BY clauses can be used to indicate secondary ordering, tertiary ordering and so on.

3.3.9 Data Manipulation

The basic elements of the Daplex syntax are statements and expressions. Statements direct the system to perform some action. FOR loops in Daplex are an example of these as well as the data definition statements already introduced. Expressions appear within statements and they evaluate to a set of entities. As an example of a query against the database described by figure 3.3 consider the query that will print the names of all persons in the database. This would be expressed as follows:

```
FOR EACH person
PRINT name(person)
```

The `FOR` statement iterates over a set of entities of type `person` and executes the `PRINT` statement for each member of the set. The complete syntax for the Daplex language is given in Appendix A. However it is worth noting that there are two basic forms of the `FOR` statement, these are:

```
FOR EACH set imperative
FOR singleton imperative
```

The term `set` refers to a set expression and `singleton` refers to a singleton expression. A set expression evaluates to a set of entities while a singleton expression evaluates to a single entity. The term `imperative` can be a `FOR` statement, an `UPDATE` statement, or a `PRINT` statement. All Daplex expressions have a value, a role and an order associated with them:

- expression value is the set of entities returned by evaluating the expression,
- expression role is the entity type under which the entities are to be interpreted,
- expression order is the ordering associated with set expressions.

3.3.10 Reference Variables

Every set expression has associated with it a reference variable. Operators which iterate over the set successively bind this variable to the entities in the set. For example in the query

```
FOR EACH person
PRINT name(person)
```

the name PERSON implicitly declares a reference variable PERSON which actually appears in the body of the FOR statement. In Daplex the reference variable could have been specified explicitly using the IN operator. Then the above query would be expressed as follows:

```
FOR EACH p IN person
PRINT name(p)
```

3.3.11 Set Expressions

A set expression is formed in one of three ways; either by a name corresponding to an entity type identifier or by a function application resulting in a set-valued result, or by the general set former construction. The form of this latter construction is:

```
sete SUCH THAT predicate
```

where `sete` is any set expression and the `predicate` evaluates to a boolean result. This set expression evaluates to those members of `sete` for which the expression is true.

3.3.12 Quantified Expressions

In Daplex, predicates following SUCH THAT can be quantified. This takes the form:

```
FOR quant set predicate
```

`predicate` evaluates to a boolean result. The term `quant` must be one of the following:

- `SOME`
- `EVERY`
- `NO`
- `AT LEAST`
- `AT MOST`
- `EXACTLY.`

Of these, `AT LEAST`, `AT MOST`, and `EXACTLY` must be followed by a singleton expression which evaluates to a boolean value.

3.3.13 Singleton Expression

These can be formed by one of the following:

- a constant literal
- a variable bound to a particular entity
- a function application producing a single-valued result
- the `THE` operator followed by a set expression

3.3.14 Aggregation

The Daplex proposals allow for aggregate functions such as `AVERAGE`, `COUNT`, `MAXIMUM`, `MINIMUM`, and `TOTAL`. To find out how many instructors there are in the department `EE`, the query would be expressed:

```
PRINT COUNT (instructor
            SUCH THAT name(dept(instructor)) = "EE")
```

The COUNT function has, as its argument, a set of instructor entities, and COUNT returns the cardinality of that set. There is a problem for aggregation in Daplex. This is due the fact that it is set based. Since set theory does not allow for duplicates, this might cause a problem when, for instance, the aggregate function AVERAGE is used. Consider the query to find the average salary of instructors in the EE department. If the set of salaries was £10,000, £20,000 and £10,000, then the query:

```
PRINT AVERAGE(salary(instructor)
            SUCH THAT name(dept(instructor)) = "EE"),
```

would incorrectly evaluate to £15000. However Daplex does provide the means to overcome this problem, using the operator OVER. The query would then be expressed:

```
PRINT AVERAGE(salary(instructor) OVER instructor
            SUCH THAT name(dept(instructor)) = "EE")
```

In this way the OVER expression evaluates not to a set but to a multiset (sometimes called a bag). A multiset therefore is a set which may contain duplicate values.

3.3.15 Derived Data

In the Daplex data model derived data refers to derived function definitions. These are functions which are introduced using the DEFINE statement. The concept of derived data means that new properties of objects are being defined based on the values of other properties. Consider the following derived function definition:

```
DEFINE instructor(student) ->>
        instructor(course(student))
```

The function `instructor` when applied to a `student` entity returns the instructors of the courses that the student is taking. The fact that this is a derived function and not a base function (specified using a `DECLARE` statement) is completely hidden from the user and the function can be used in the same way as if it was a primitive function. Derived functions can also be specified over system-supplied entity types, such as `STRING`, `INTEGER` and `BOOLEAN`. The declaration:

```
DEFINE student(STRING) -> INVERSE OF name(student)
```

will map a given string into a set of students. The derived data mechanism is very powerful. It allows for the creation of user views. In addition to the operators already introduced, the following operators are available:

- `TRANSITIVE OF` - an example definition would be

```
DEFINE superior(employee) ->>
        TRANSITIVE OF manager(employee)
```

The specified function returns the set containing the manager of the employee. the manager's manager and so on.

- `INTERSECTION OF`, `UNION OF`, `DIFFERENCE OF` - these are used to form set intersections, unions and differences. As an example of using one of these, consider the query:

```
DEFINE student-teacher( ) ->>
        INTERSECTION OF student, instructor
```

This will create a new entity type student-teacher using primitive entity types.

- COMPOUND OF - this creates derived entities corresponding to the elements of the Cartesian product of its operands. The following example illustrates usage of this operand:

```
DEFINE enrolment( ) ->>
        COMPOUND OF student, course(student)
```

This declaration defines the set of `enrolment` entities, the `enrolment` entity type. In addition it implicitly defines the two functions `student` and `course` to operate over entities of type `enrolment`.

3.3.16 Constraint Specification

There are two aspects to constraints specification. Firstly a means is provided to prevent updates taking place against the database which would violate certain system imposed conditions. For example, consider the constraint that the head of a department must come from within that department. This would be specified as follows:

```
DEFINE nativehead(department) ->
        dept(head(department) = department
```

For those departments that satisfy this constraint, this function will evaluate to true, while for those that do not it will evaluate to false. The Daplex syntax provides for the use of the keyword CONSTRAINT, as follows:

```
DEFINE CONSTRAINT nativehead(department) ->
        dept(head(department) = department
```

This causes the system to abort any update transactions which evaluate to false. In addition constraints may be specified over the database as a whole. For instance, there might be a constraint that the number of managers in the database must not be exceeded by the number of non-managers. Such a constraint could be specified as follows:

```
DEFINE manager( ) ->> manager(employee( ))

DEFINE nonmanager( ) ->>

      DIFFERENCE OF employee, manager

DEFINE CONSTRAINT toomanychiefs( ) ->

      COUNT(manager( )) < COUNT(nonmanager( ))
```

The second aspect of constraint specification is concerned with the `TRIGGER` facility. The facility allows for the execution of a specified imperative when the function over which it is defined changes from false to true. Consider the situation where the department head needs to be informed when more than 40 students are enrolled in a class. This could be done as follows:

```
DEFINE student(class) -> INVERSE OF class(student)

DEFINE TRIGGER overbooked(class) ->

      COUNT(student(class)) > 40

      SENDMESSAGE(head(dept(class)),
                        "OVERBOOKED:", title(class))
```

3.3.17 Database Update

In Daplex, update statements specify the value returned by a function when that function is applied to particular entities. The syntax involved will be illustrated by a number of examples. For example, to add a new student named `Helen`, whose department is `CS` and courses taken are `Operating Systems` and `Compiler Construction` to the database:

```
FOR A NEW STUDENT
    BEGIN
        LET name(student) = "HELEN"
        LET dept(student) = THE department
            SUCH THAT
            name(department) = "CS"
        LET course(student) =
        (THE course SUCH THAT name(course) =
            "OPERATING SYSTEMS",
            THE course SUCH THAT name(course) =
            "COMPILER CONSTRUCTION")
    END
```

3.4 CONCLUSIONS

Suppliers of DBMS have increasingly used Unix as a means of reaching large numbers of users. Many products have been developed to run under Unix, while older ones have been adapted to do this. Unix is naturally suited to database applications for several reasons. Firstly it is a multi-user operating system and can therefore handle typical database environments in which information is a common source. Secondly, from DBMS suppliers' point of view, Unix provides a considerable degree of software portability between hardware from different manufacturers. Thirdly, because DBMS for Unix systems are written in the high-level C programming Language, they can easily be transferred from one Unix system to another. From a commercial point-of-view, the growing popularity of Unix means that such portability gives suppliers a potentially large user base. Since Unix systems have only recently achieved commercial success, systems software products for them reflect the latest technical advances. In terms of DBMS this has meant an emphasis on relational databases. The above mentioned advantages have lead to the selection of the Unix operating system as a suitable environment for the implementation of the research system.

Ingres was selected as the target DBMS due to the fact that it is a relational DBMS and is widely available in Unix and other environments. The significance of this is that the portability of a system if it is based on Ingres, is increased. Further, its use provides the advantages of relational databases. Furthermore, Quel is its query language. Quel was selected as a suitable target query language because it is based on relational calculus and is therefore relationally complete and its operations are based on relations.

The Daplex functional model proposed by Shipman has semantic expressiveness as its major advantage. This is due to its inherent ability to organise data into type hierarchies with automatic inheritance of attributes and relationships from a supertype to all of its subtypes. For example, entities are organised into entity types which in turn are organised into subtype-supertype hierarchies. Properties of entities are represented by functions. Extensions of entities can belong to different entity types, which bears a correspondence to the different roles that objects can play in the real world. Another advantage of this model is the fact that no distinction is made between the properties of entities and relationships between entities. They are both modelled as functions. Further the model provides a complete set of operations and provides the facilities to specify constraints and derived data. The advantages outlined above lead to the selection of this model as a suitable basis for this implementation.

For these reasons the practical work for this research was an implementation of a Daplex front-end to the Quel target language on the Ingres DBMS running under Unix. This front-end system (FQLFE) is described in chapter 4.

CHAPTER 4

THE FRONT-END SYSTEM

4.1 INTRODUCTION

The data modelling capabilities of Daplex incorporate those of the hierarchical, relational and network models. This increases its suitability as a front-end for existing database systems. However, since the relational model, used as the 'target model', is a subset of the functional model, any Daplex description of a relational database will be subject to the following limitations:

- no multi-valued functions are allowed
- functions cannot return user-defined entities
- multiple-argument functions are not allowed
- there are no subtypes.

Having specified the Daplex description with the existence of a suitable front-end, Daplex requests can be written against the relational database. By only specifying the relational database in terms of Daplex, the full benefits of the Daplex approach will not be available because of the limitations (specified above) of the underlying data model. A functional view would be more useful and this can be done (by defining derived functions) using FQLFE. With these additional definitions, queries may be written in Daplex against the relational database, as if it were a Daplex database. This specification of the functional view consists of adding semantic information which was not expressible in the relational data model.

The Functional Query Language (FQLFE) system is an attempt to provide such a front-end. Its overall structure is shown in figure 4.1. This chapter discusses the main features of the FQLFE implementation.

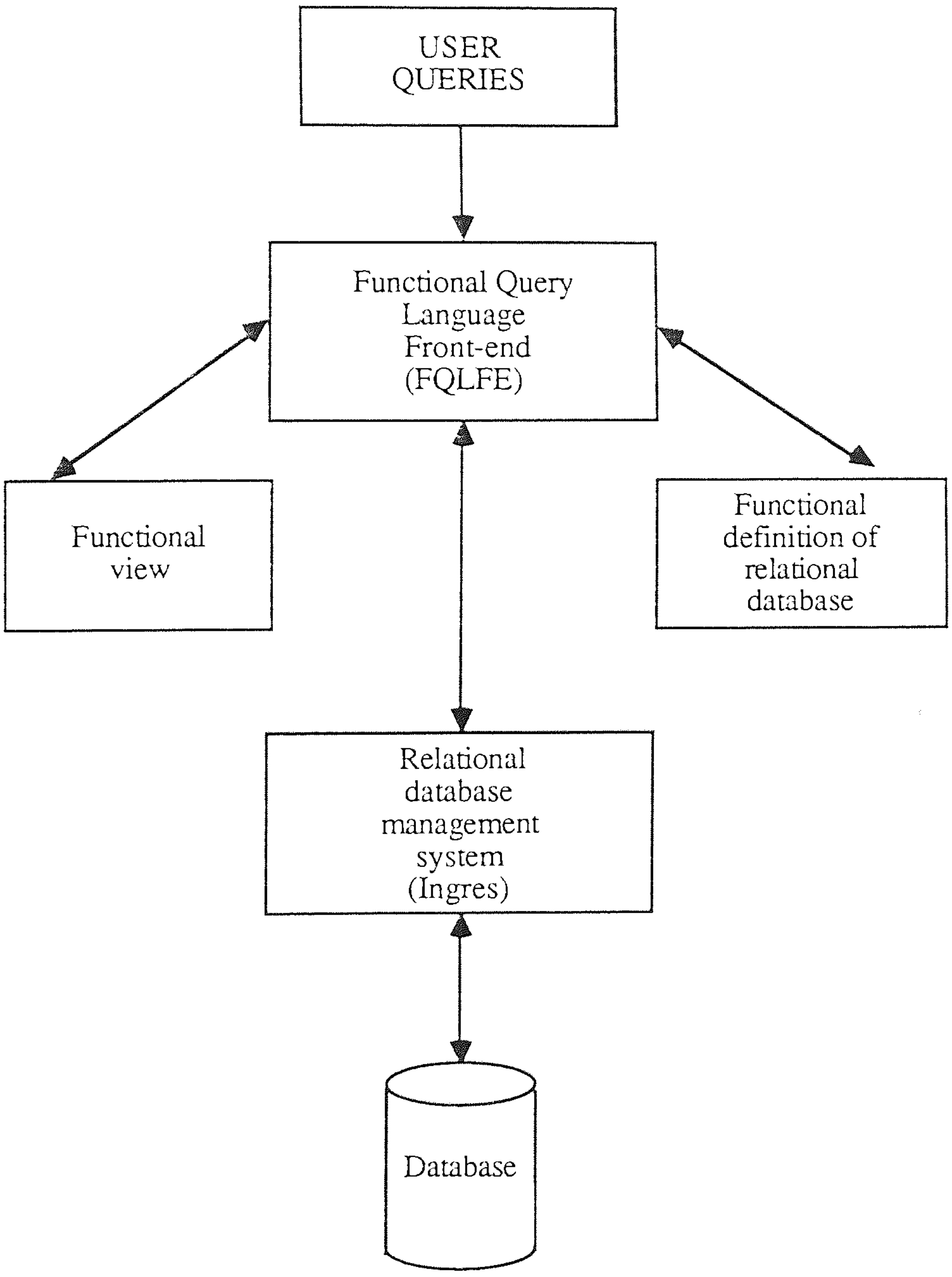

USER QUERIES
Functional Query Language Front-end (FQLFE)
Functional view
Functional definition of relational database
Relational database management system (Ingres)
Database

Figure 4.1: System Overview

4.2 SYSTEM INVOCATION

To execute the FQLFE system, the user types in the following command:

```
fqlfe <dbname>
```

where <dbname> is the name of an existing database. When the system is invoked, it verifies the existence of the relational database `<dbname>`. The catalogue component of the system is invoked if a valid database name is specified. This checks the user's home directory for the existence of a local FQLFE directory called `.FQLFE`, which, if it exists, will contain any information regarding the user's functional view of the relational Ingres database. If such information exists, it is loaded into main memory. The local FQLFE directory will be accessed and updated as necessary during the user's database session. If the necessary information does not exist, that is, the user does not have a functional view for accessing the database, then this program calls the necessary routines to set up the basic information. Initially, the relational database system is accessed and information regarding the database is retrieved. This information consists of the names of the relations in the database, their attributes, and the types of the attributes. In the beginning of the session this information is resident in main memory, but will be written to a disk file at the end of the session. Figure 4.2 shows the data structure used to hold information about the relational database in main memory. The disk file `<dbname>.rel` is used to hold information about the relational database, while the diskfile `<dbname>.func` is used to hold information about the functional view.

If a file containing the relational information about the database does not exist, then it is assumed that the database also does not have any functional view information stored. In this case the relevant information will be generated during the database session. At the end of the session this information will be written to a disk file, called `<dbname>.rel`, for future reference. Figure 4.3 shows the structure of the file `<dbname>.rel`. The total number of relations in the database is specified by the value `i`, while the number of attributes in relation `i` is specified by `ji`. Details of the structure of the file <dbname>.func are given later in this chapter.

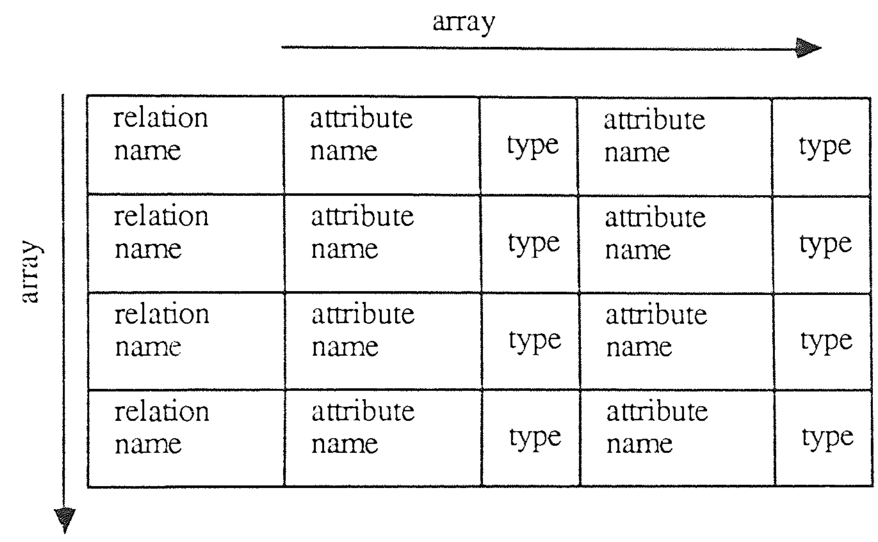


Figure 4.2: Catalogue Information Data Structure

```
i
relname1
j1
attribute1 attribute2 attribute3.....attributej1
relname2
j2
attribute1 attribute2 attribute3.....attributej2
......................
.......................
relnamei
ji
attribute1 attribute2 attribute3.....attributeji
```

Figure 4.3: File Structure for Catalogue Information

4.3 MONITOR

The interaction with the user is based on the workspace concept. The idea is that user commands to query the database are written into a workspace where they can be manipulated in various ways. `Monitor` commands are used to manipulate the workspace and its contents. The workspace is implemented as a disk file called

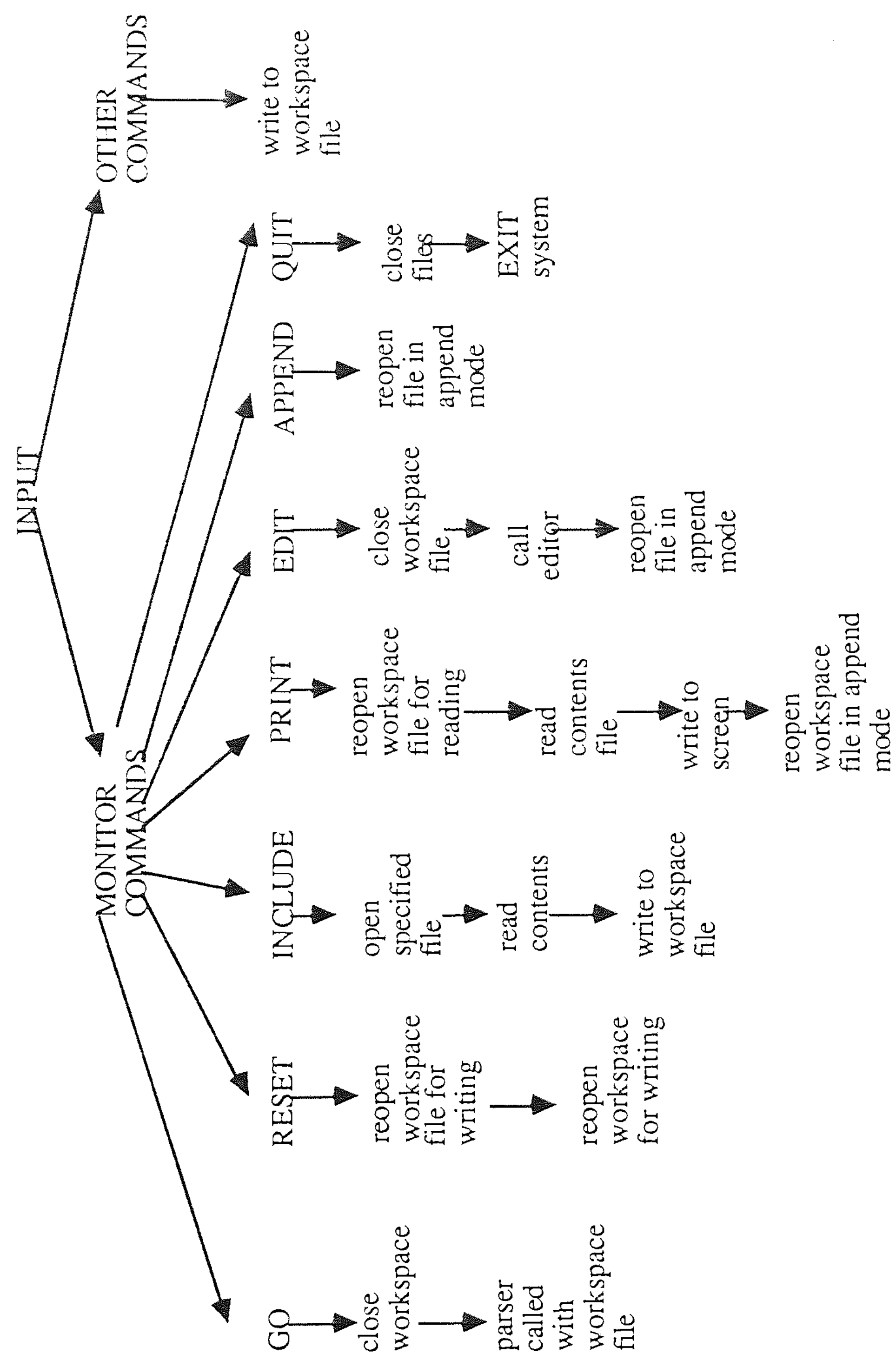


Figure 4.4: Input Handling

`FQLFE_workspace`. The `monitor` handles all user input and divides it into two types. The first type is its own commands, these have the syntax:

```
\<command>
```

where <command> is one of the following keywords:

- `go`
- `reset`
- `include`
- `print`
- `edit`
- `append`
- `quit`

The second type is any commands that are not of the above form. These are written to the file `FQLFE_workspace`, on the assumption that they are query language statements. Each of the `monitor` commands has a particular function and causes a number of different actions to take place (see figure 4.4). These are summarised below.

- `\go.` When this command is entered, the file `FQLFE_workspace` is closed. The `monitor` program generates a child process which executes the language recognition component of the system. For the child process, the input is redirected so that file `FQLFE_workspace` is read. This file represents the user's workspace and contains the user's query language statements. Therefore the affect of this command is to attempt to execute the user's database queries/updates.

- `\reset.` This command is used to reset the contents of the user's workspace. Thus it erases any previous statements entered by the user which are currently in the workspace. This is implemented by closing the file `FQLFE_workspace,` which represents the user's workspace and reopening it for writing. The file is reset to size zero bytes and its previous contents are lost.

- `\include`. The complete syntax of this command is :

```
\include <filename>
```

  This command is used to load prewritten statements from another existing file into the system workspace. The contents of the file specified by file `<filename>` are loaded into the file `FQLFE_workspace`. Any previous contents of the workspace are overwritten. This command allows frequently used sets of commands to be written to a file so that they can be loaded up when required, rather than repeatedly typing in the commands directly.

- `\print`. This command is used to print out the current contents of the workspace on the screen. This is achieved by closing the file `FQLFE_workspace`, opening it for reading, and writing its contents to the screen. It is then closed, and reopened for appending.

- `\edit`. This command is used to edit the contents of the workspace. If a command is typed in incorrectly, the user can make alterations to it rather than type in the statements again. The editor that is made available to the user is the Unix screen editor *vi*. Changes, if required, can be made to the commands in the workspace using the editor *vi* in the normal way. At the end of the editing session the changes can either be abandoned or written into the workspace. This edit command is implemented by the monitor process generating a child process which executes the *vi* command, and the file specified is `FQLFE_workspace`.

- `\append`. This command is used to add statements to the commands currently in the workspace. Normally after a statement is executed, and another statement is typed in, the workspace is reset. The effect of this command is to override this feature, if the user requires it. It is implemented by closing the file `FQLFE_workspace` and reopening it in append mode.

- `\quit`. This command is used to exit from the FQLFE system.

After execution of any of the `monitor` commands, control is always returned to the `monitor`. This puts up its prompt on the screen and awaits further user input.

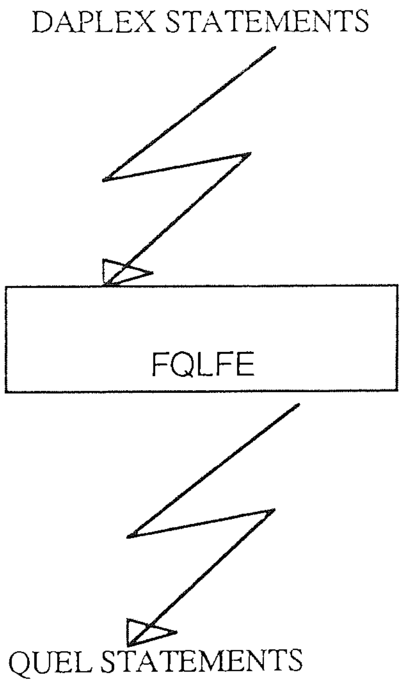


Figure 4.5: Basic FQLFE function

4.4 DATA DEFINITION AND MANIPULATION.

The basic function of FQLFE is to transform Daplex data definition and manipulation statements into Quel data definition and manipulation statements (see Figure 4.5). This is not possible in one single step. In FQLFE, this function is split into several steps, which are shown in figure 4.6. Each of these steps has certain expected inputs from which it generates its output. This output forms the input to the next step until the target language statements are arrived at.

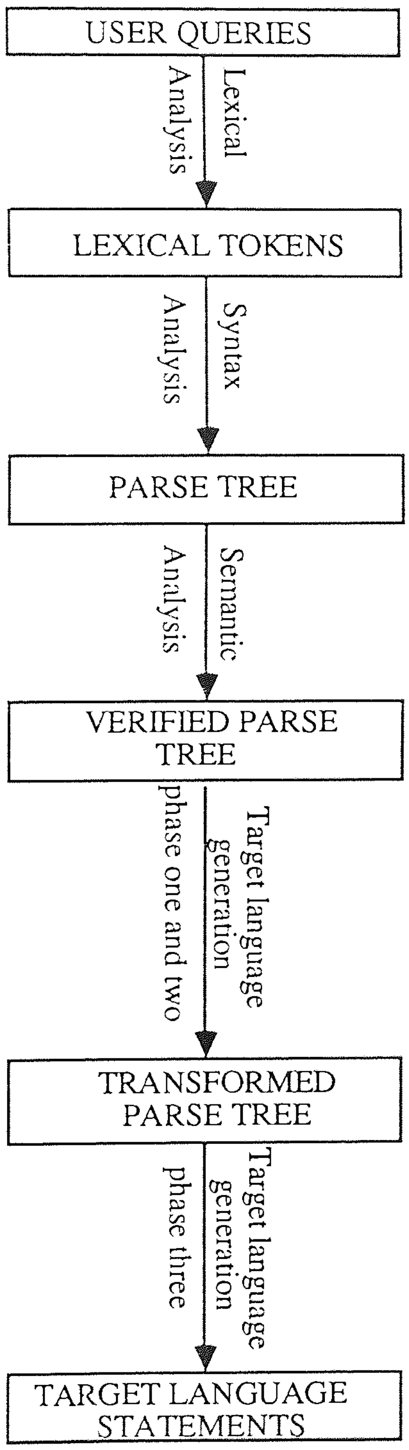


Figure 4.6: Query transformation

4.5 LEXICAL ANALYSIS

The lexical analyser was generated with the aid of the Unix tool Lex (Lesk & Schmidt, 1978), which is a lexical analyser generator. The `monitor` passes the file `FQLFE_workspace`, which contains all the user input that did not consist of `monitor` commands. This happens when the command `\go` is entered by the user. The input passes through the lexical analyser and is broken down into various tokens according to the rules specified in the lex program (see figure 4.7).

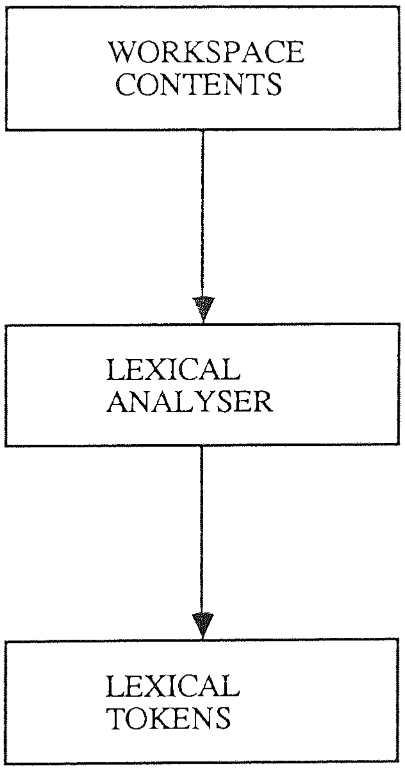


Figure 4.7: Lexical analysis

The Lex specification for the FQLFE system is given in Appendix B. The input is divided into various defined tokens. Any input that does not correspond to an expected pattern generates a special token. This is passed to the next phase of the system in the same way as the other tokens, and does not cause an error message to be generated during the phase. By detecting and reporting the error during syntax analysis, the user can be given more information about the nature of the error. The output from the lexical analysis phase forms the input to the syntax analysis phase.

4.6 SYNTAX ANALYSIS

This component was developed with the aid of the Unix tool Yacc (Johnson, 1978), which is a tool for developing compilers. The input to this part of the system consists of the tokens generated by the lexical analysis. The tokens are processed according to a set of rules which form the Yacc specification (see figure 4.8). This specification is a grammar which defines the language Daplex. The Yacc specification used in this implementation is given in Appendix C.

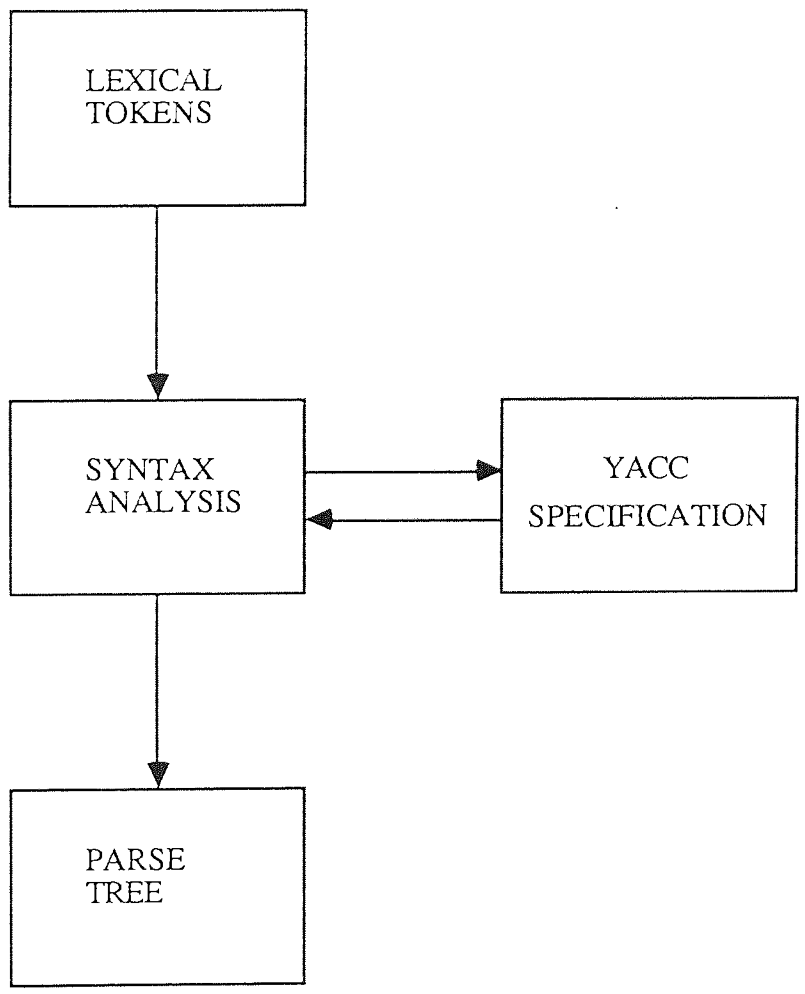


Figure 4.8: Syntax Analysis

During language recognition a tree of the input is built up. If an error is detected, a relevant message is put up on the screen to inform the user. The mechanism used for building a syntax tree of the input is based upon having a structure for each rule of the Yacc specification. For example, consider the following rule:

```
declarative         :     simple_decl

                    |     complex_decl

                    ;
```

This rule would have the following structure described for it:

```
STRUCT declarative_type

{

     int type;

     UNION

     {

          STRUCT branch_declarative_1

          {

               SIMPLE_DECL_TYPE  simple_decl_1;

          }BRANCH_declarative_1;

          STRUCT branch_declarative_2

          {

               COMPLEX_DECL_TYPE complex_decl_1;

          }BRANCH_declarative_2;

     }RULE;

};
```

The Yacc specification for syntax analysis is extended to include the following:

```
declarative        :     simple_decl
                         {
                                 $$=(DECLARATIVE_TYPE)
                                       node2(1,$1);
                         }
                   |     complex_decl
                         {
                                 $$=(DECLARATIVE_TYPE)
                                       node2(2,$1);
                         }
                   ;
```

Therefore, when a declarative is found, the routine `node2` is called. If a `simple_decl` is found, a value of 1 is passed to the routine. This specifies the type of `declarative` found. In this way, if a `complex_decl` is found, the value for type is set to 2.

There is a number of node routines. There is a different routine according to the number of parameters being passed. For the purpose of the system written as part of the research, the following node routines exist:

```
node2, node3, node4, node5, node6, node7.
```

The routine, `node`*n*, is passed the relevant number of parameters (*n*). In every case, the first parameter is the type value. This type value will vary according to the particular alternative found within a rule. The remaining parameters represent the symbols making up the rule. They are pointers to structures.

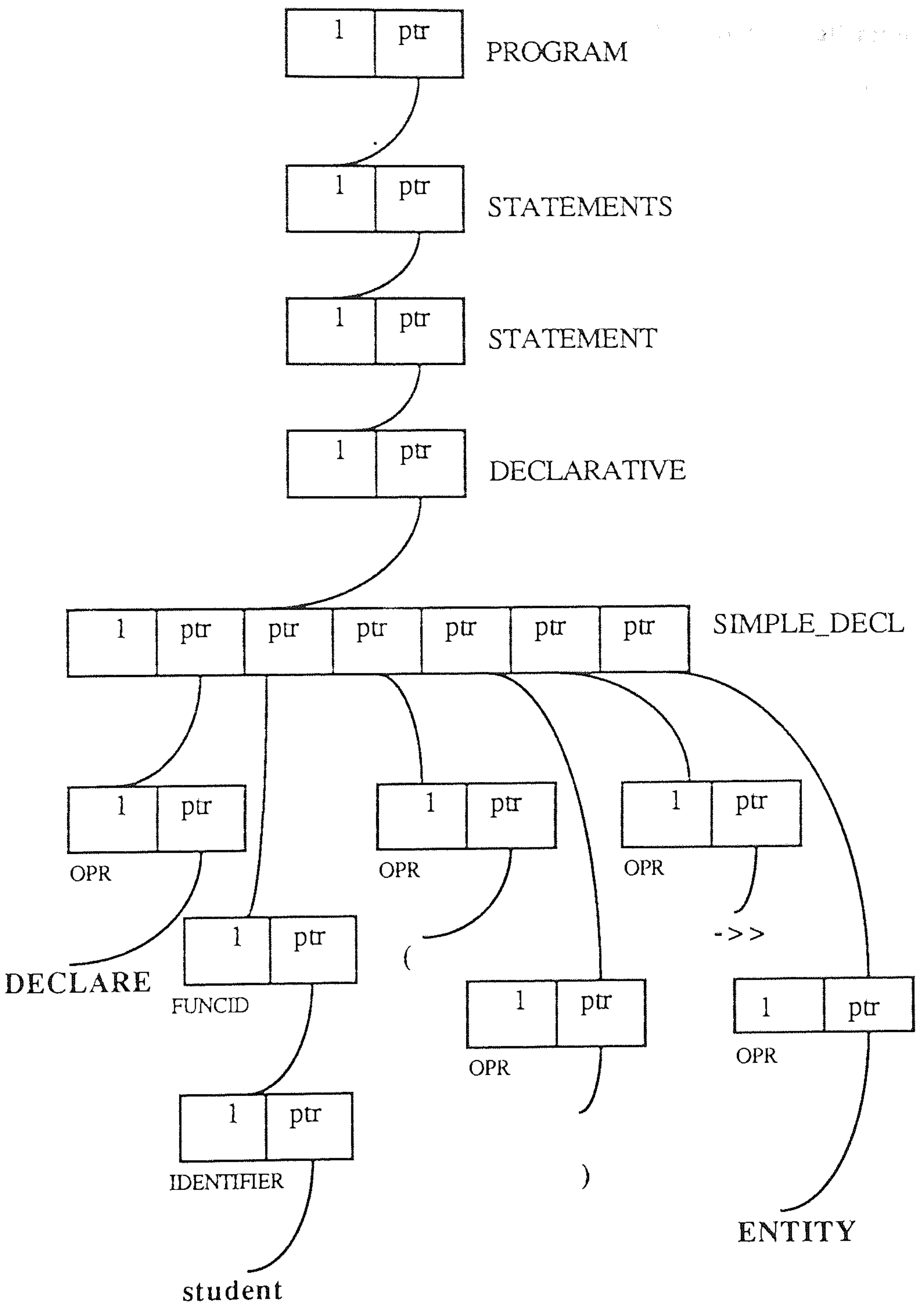


Figure 4.9: An Example Parse Tree

The routine `noden` calls the standard C library routine *malloc* to allocate memory in order to store the parameters. The amount of memory allocated will depend on the number of parameters. For example, for the routine `node4`, it will allocate a block of memory large enough to hold four pointers to integers. Therefore, for *n* parameters, it will allocate memory to hold *n* pointers to integers. The routines all return a pointer to a character type. This pointer points to the allocated node. In the action:

```
$$=(DECLARATIVE_TYPE) node2(1,$1)
```

the C cast operator, `(DECLARATIVE_TYPE)` forces the pointer to point to a structure of type `DECLARATIVE_TYPE`. In this way, pointers are returned bottom upwards, and a syntax tree is formed of the language statements. Figure 4.9 shows the syntax tree that would be generated for a simple statement such as:

```
DECLARE student( ) ->> ENTITY
```

The node allocated will vary according to the number of symbols in the righthand side of the rule that need to be stored. Pointers to these nodes are passed back upwards as the tree is climbed. On completion of the syntax analysis, a tree exists in main memory which represents the input.

4.7 SEMANTIC ANALYSIS

During semantic analysis, the parse tree that was generated during syntax analysis is processed. This is achieved by passing to this step the pointer to the root node of the tree. The semantic analyser traverses the tree, checking that the statements input by the user are semantically correct. The syntax analysis phase only ensures that the input was syntactically correct. It is possible that syntactically correct input is semantically incorrect. During this phase there are two types of semantic checks carried out. These are:

- To ensure that the Daplex language is correct.
- To ensure that the input statements are correct in terms of the underlying relational database.

4.7.1 Data Definition - Relational

During data definition, a base entity can be defined, for example using the following:

```
DECLARE student ( ) ->> ENTITY
```

The identifier `student`, names:

- the entity type,
- the set of person entities and
- the function which produces the set.

The identifier specified must be checked to ensure that it corresponds to the name of a relation in the relational database. The table of catalogue information is accessed for this purpose. If a corresponding relation does not exist a relevant error message is displayed to the screen and no entry is made into the table of base functions. If it is a valid entity, the relevant entry is made in the base function table. In a declaration such as:

```
DECLARE name(student) -> STRING
```

The identifier `name` names:

- the function which is an attribute of the argument `student`.

In addition, `student` must be a base function, that is, it is an entity in the functional view of the database. This will also ensure that there is a corresponding relation in the relational database. This relation also needs to be checked, to ensure that `name` is an attribute of the relation `student`. In addition, the type `STRING` being declared means that the attribute in the relational database should be of type character. In a declaration such as:

```
DECLARE teaches(course) -> instructor
```

The identifier `teaches` names:

- the function which represents the relationship between the identifier `instructor` and the argument `course`.

In this case, the checks will need to be made as specified in the previous declaration, except that the return-type `instructor` needs to be checked, to ensure that it is a valid entity type for the functional database. This can be confirmed by accessing the base function table. For valid declarations such as the above, entries are made in the non-base function table.

4.7.2 Data Definition - Functional

A simple example would be a statement such as:

```
DEFINE dept(student) -> department SUCH THAT
        deptno(department) = deptno(student)
```

The verification that needs to be done in this case is as follows. The entity type `department` must be a previously defined base function. Its existence should be verified by accessing the base function table. This is also true of `student`. In terms of the database, `department` must be a relation of the relational database, it must also have an attribute `deptno`. In a statement such as :

```
DECLARE course(instructor) ->>
        INVERSE OF instructor(course)
```

the `instructor` function must exist as a base function. This means that it must also exist as a relation in the relational database. In both of these cases, the named argument must be an entity, that is, it must have an entry in the base function table.

This also means that a corresponding relation exists in the relational database. The identifier naming the return-type should either be a data type or an entity.

4.7.3 Data Manipulation

During data manipulation, verification checks are made on the base function table and the non-base function table. No checks need to be made against the database, since queries can only be made against the specified functional view of the relational database. For example, consider the query:

```
FOR EACH student  SUCH THAT

FOR SOME course(student)

        name(dept(course)) = "EE"

PRINT name(student)
```

For this query, the function tables must be checked to ensure the existence of the functions `course`, `name` and `dept`. The function `course` must accept arguments of type `student`. The function `name` must accept arguments of type `dept`, while the function `dept` must accept arguments of type `course`. In addition, the return-types of all the functions must be verified. The `course` function should return a `name` type, while the `name` function should return a `STRING` type.

4.8 TARGET LANGUAGE GENERATION

When a verified tree of the input has been built up, one of two alternative actions take place. If the user's input statements are data definition statements, then the relevant function table is updated with this information. Alternatively, if the input statements are data manipulation statements, then the parse tree must be traversed and syntactically transformed to produce the relevant Quel language statements.

The target language generator is passed the pointer to the root node of the parse tree, which it will traverse. This module consists of routines which will traverse a parse tree generated for any user input which the Yacc specification and actions has produced. As the tree is traversed, the following syntactic transformations take place:

- The parse tree is scanned, and any derived function calls are replaced by the syntax tree representation of the return expression. This phase generates another tree.
- This new tree is scanned and all the FOR SOME statements (that is, `imperative` statements) are collected together.
- Each of the expressions for the FOR SOME statements are linked with the AND operator.
- Every FOR EACH statement and FOR SOME statement is replaced by Quel `range` statements.
- The transformed expressions are used to produce Quel statements.

4.8.1 Phase One Transformations

The pointer to the root node of the syntax tree of the query statements is passed to the phase one transformation routines. This phase involves traversing the syntax tree in the same way as the tree was traversed during semantic analysis. During phase one, the derived function table is accessed to check whether the particular function names specified are derived functions. If they are, a check is made as to whether the function call is part of the FOR statement specification or whether it occurs in the body of the FOR. The place where the function call occurs determines how it is treated. For example, in the query

```
FOR EACH employee SUCH THAT

      name(dept(employee)) = "toy"

PRINT name(employee)
```

the function `dept`, is a derived function, whose definition is:

```
DEFINE dept(employee) -> department SUCH THAT

      deptno(sales) = deptno(department)
```

In theory, different functions can have the same identifier. For example, there can exist a function `name` which, when applied to an `employee` entity returns the name of an employee and when applied to a `department` entity returns the name of the department. Both of these functions have the same name but actually refer to two different functions. Therefore, when accessing the derived function table to check the name specified in a function call, it is not sufficient simply to compare the called function name with the table function name. The arguments to the functions also need to be compared as this will distinguish whether or not it is the same function. This is not a simple problem to address, since derived functions can have the following syntax:

```
funcid( mtuple) ->...........
```

An effective way to solve this problem is to call the routine `output_mtuple` from the `output` module, for the stored function and pass to it the relevant pointer. This will then write the syntax tree for `mtuple` to a temporary file. This file is then accessed and the contents read into a string. The routine `output_mtuple` is then called for the function call and the above procedure repeated. The two strings are then compared to see if they refer to the same function. Only then can it be confirmed that the called function corresponds to the one in the derived function table.

In the above case, when `dept` has been recognised as a derived function, the return-type department, is substituted for the function call - the nested function call `name(dept(employee))` becomes `name(department)`. This is carried out by the pointer to the function call in the syntax tree, being altered to point to this return-type. The right-hand side of the function definition contains information necessary to transform this query into Quel, It must also be included in the syntax tree at this stage. The important factor to consider when making any changes to the syntax tree is that regardless of any changes being made to incorporate the transformations, it must still retain a format that allows traversal for phases two and three in the same way as phase one. Thus, these routines should traverse the abstract tree so that, for any actual tree, traversal using the same routines is possible. A simple, but effective way of incorporating this definition into the transformations is to use the list `deflist`. This list consists of nodes where each of the nodes has the structure shown in figure 4.10. The field `defptr` is set to point to the return-type for this function definition. This list will be used again at a later stage. Nodes of this routine are created by using the C library routine *malloc*. The `next` field of the current node is set to `null`. The previous node in the list is set to point to this newly allocated node. The head of the list consists of a node which points to the first item in the list. It does not have a value in its `defptr` field. In addition, an entry will be made into another list, `linklist`, which has the structure shown in figure 4.11 The address of the function call node is stored in the allocated node of the linklist. The link field has the address of the conditions which form part of this function's definition stored in it. Therefore the effect is to replace the derived function call by its return-type.

A derived function call which occurs in the FOR specification as in the query:

```
FOR EACH item SUCH THAT
FOR SOME sold(item)
     floor(department) = 2
PRINT name(item)
```

results in different actions being carried out. Instead of replacing the function call by the derived function return-type and adding the definition to deflist, the derived function call is replaced by all of the right hand side of its definition. The pointer to this value is obtained from the derived function table. Since the extra semantic information needed to transform the Quel query has all been incorporated into the query, no node is added for this to deflist.

4.8.2 Phase Two Transformations

In this phase the routines transform_imperative and transform_forloop include the following actions. In transform_imperative, the lists rangelist, forlist, predlist and condlist each have a pointer initialised to point to the head of the list. The head nodes are the only nodes currently present in each of the lists. For this reason, the next pointer in the node at the head of each of the lists is set to null.

The forlist is used to point to all FOR statements. These either begin with FOR EACH or FOR SOME. The condlist is used to point to all statements that are gpimperative statements, that is, they begin with PRINT. When an item has to be added to these lists, a node of the relevant type is created using *malloc*. The previous node in the list has its pointer next set to point to the address of this new node. The relevant pointer, (predptr, forptr or condptr) is set to point to the syntax tree of the statement to be added to the list. The pointer next of the new node is set to null.

FOR statements can also occur within other types of statements, for example, within `pred` clauses (see Appendix A). Therefore the routine `transform_pred` also contains the code to add an item to the `forlist`. It is in this routine that items are added to another list, `predlist` (see Figure 4.10). This has exactly the same structure as `forlist`, `condlist` and `deflist`, and items are added to it in the same way. Essentially, phase two consists of traversing the syntax tree in main memory and putting pointers to different types of clauses into nodes in the relevant lists. The various lists generated will be used during phase three.

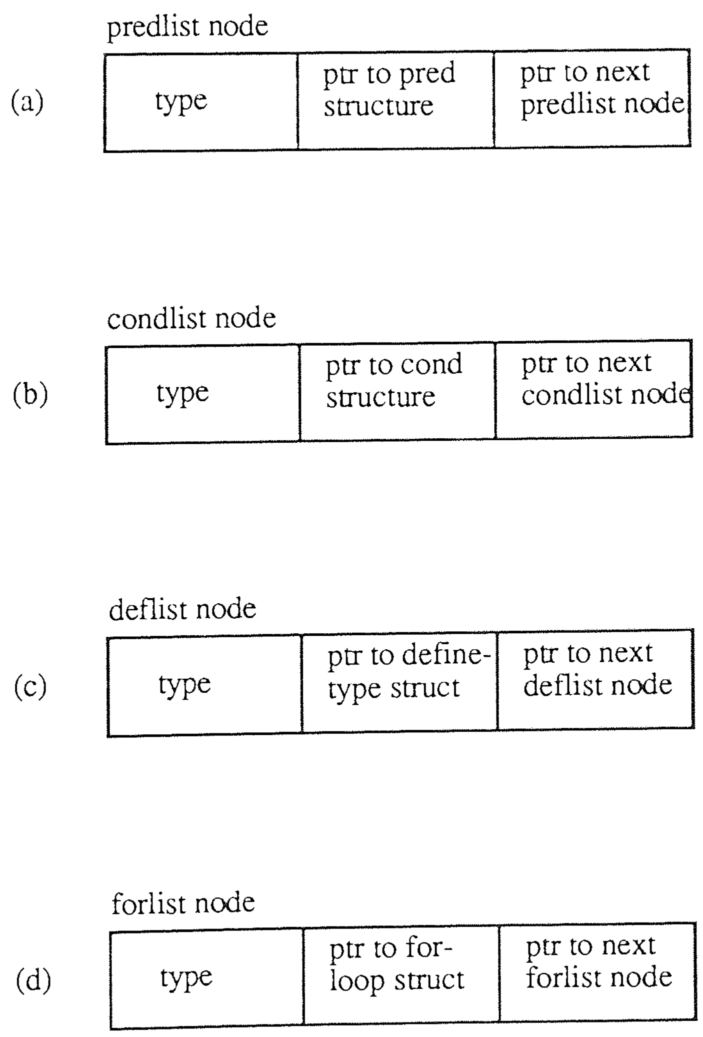


Figure 4.10: Node structures of deflist, condlist, predlist and forlist.

4.8.3 Phase Three Transformations

To finally arrive at the Quel transformation of the Daplex query, the tree of the query, held in memory, is traversed. In this instance, however, the traversal does not necessarily start at the root node. Traversal starts at the first item in the `forlist`. Depending on the type of item in a particular node of `forlist`, either the routine `transform_forloop` or the routine `transform_pred` is called. As each node of the list is processed the relevant routine is called.When the list is exhausted, that is, when the pointer `next` is `null`, the next list is processed. All nodes in `forlist` represent `FOR` statements, which will be transformed into corresponding Quel `range` statements. The range variables must be allocated by the FQLFE system in a systematic way. This is done by adding the name of the relevant set into the list `rangelist`. This list has the structure shown in figure 4.11.

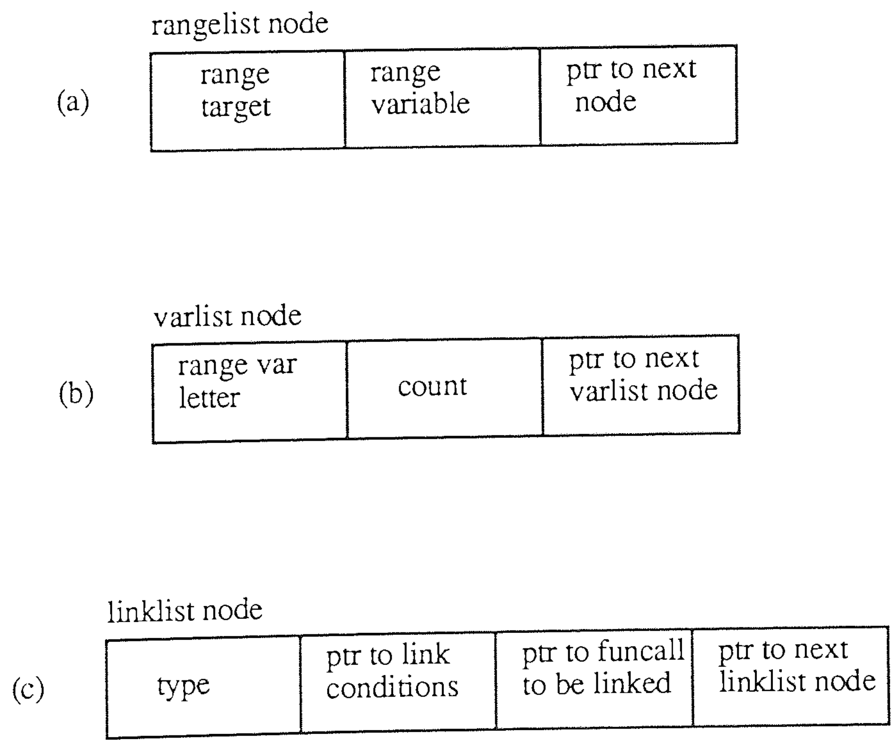


Figure 4.11: Node structures of linklist, rangelist and varlist

Range variable allocation is carried out as follows. When an identifier specifying the set over which the variable is to range is encountered, this corresponds to a `typeid`.

The rangelist is accessed to see if that identifier already exists in the list, and if it does not it is added to the list. The list of range variables is checked to see if the first letter of this identifier is already in existence as a range variable. If it is, then the count of the number of variables using this letter is incremented. The item in the `rangelist` therefore has associated with it a range variable consisting of the first letter, for example, `s`, or the first letter and the value of the count for that letter, for example, `s2`. In this way, range variables allocated are always unique even if they reference a set which begins with the same letter as another set or if they range over the same set.

The next list to be processed is `deflist`. This was generated as a result of incorporating any derived function calls into the query. If this has any items in it, they are processed by calling the routine `transform_definetypes`, and the pointer in the `defptr` field is passed to it. Again, the transformation of each item in this list results in a Quel `range` statement. Range variables are allocated as detailed above.

When `deflist` is exhausted, `condlist` is processed. All items in `condlist` are Daplex `imperative` clauses. To transform these, the routine `transform_imperative` is called with the pointer `condptr` passed to it as a parameter. Each item held in this list corresponds to a Quel `retrieve` statement. As this list is transformed, the target-list to be retrieved by the Quel query is built up.

Finally, `predlist` is processed. All of its items are of one type, and, for each item `transform_pred` is called with the relevant pointer passed to it. Items in `predlist`, correspond to the Quel `qualification` statements.

As a result of the ordered processing of these list structures, the transformed Quel query is produced in the format:

```
range statements...
.....
.....
retrieve (.............)
where
.............
.............
```

During list processing in phase three, routines which produce ouput write this to the file `FQLFE_query`. The child process which accesses the database, is generated by means of the Unix *fork* utility. The database access program sets up the environment, so that Ingres reads its input from this target language file rather than from the terminal.

4.9 DATABASE ACCESS

The routine `dbexec` is called when access to the Ingres database is required. A child process is generated. In this process the input is redirected so that instead of reading from the terminal, input is read from the relevant file. During catalogue creation, this routine is called with the file `catalog_request` as input and the file `result` as output. After query transformation, when the target Quel statements have been generated, the same routine is called but this time with the file `FQLFE_query` as input. This contains the FQLFE-generated Quel query. This file was produced by transforming the user's Daplex query which was contained in `FQLFE_workspace`. The forked process is overlaid by the Ingres process and the name of the database that the user is currently querying is passed to it as a parameter. As well as input being redirected to read from a file, a very simple but effective mechanism is used to prevent the Ingres process writing its output to the screen. Its output is also redirected to a file (see Figure 4.12). After execution of the database commands, this file is accessed and the relevant information selected.

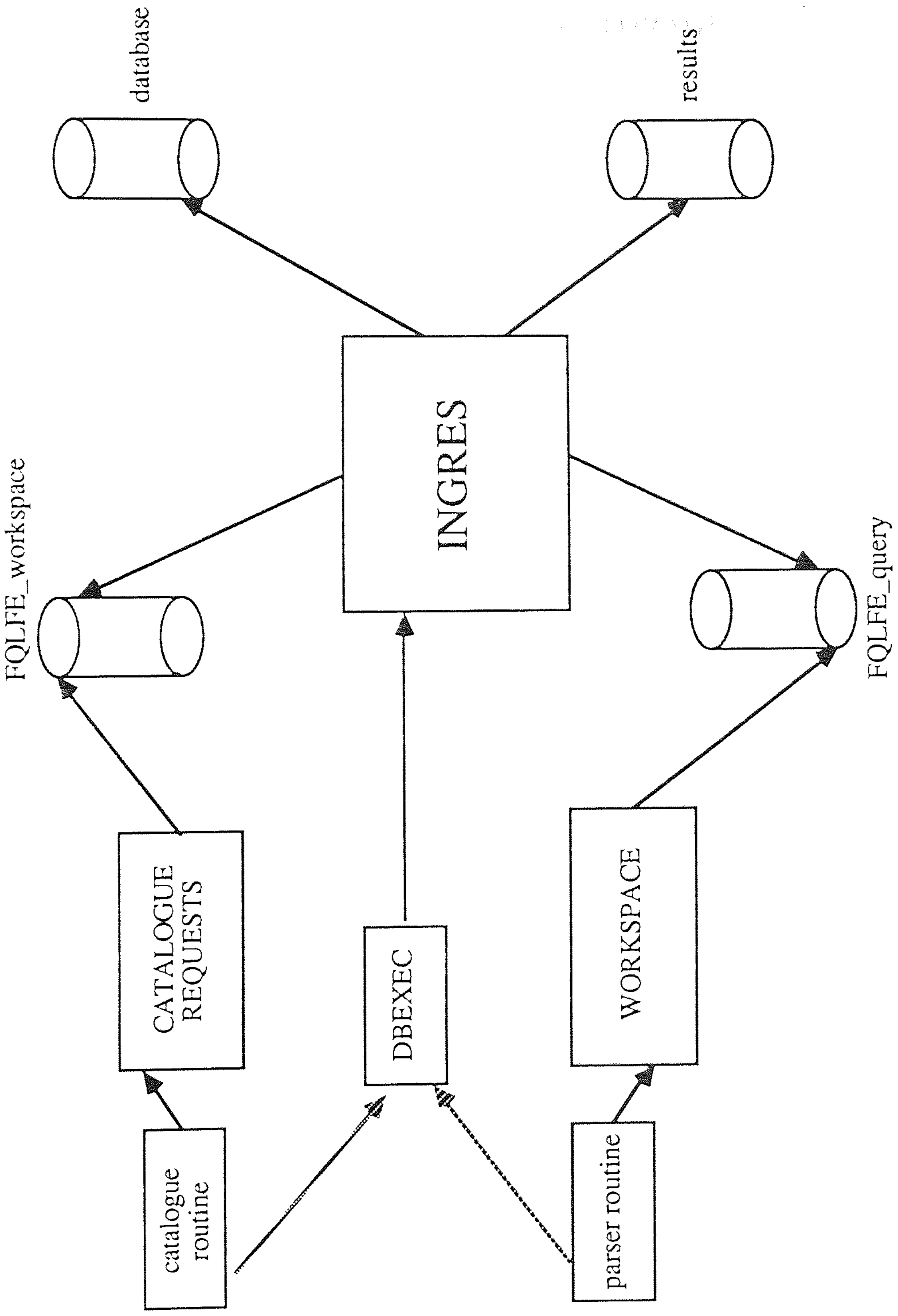


Figure 4.12: System Interaction With Ingres

4.10 DATA DEFINITION - INFORMATION STORAGE

Details of how Daplex queries are transformed into Quel queries have been given. This is not the only action that can result. If the user has provided data definition statements, these will not be transformed into Quel statements. Instead various system tables are updated. Information about base, nonbase, and derived functions is held in three main tables. These tables are set up internally as linked lists, rather than as arrays. The advantage of this is that storage space is not allocated unnecessarily. When a function needs to be added to any of the tables, the relevant node for the table is allocated. This is then inserted into the list by setting the pointer in the previous node to this new node and setting the `nextfunc` field of this new node to `null` (to indicate the end of the list). The structure of these lists is given in figure 4.13.

In addition to the above tables, the following information is also stored:

- basetot - the total number of base functions for this database
- nbtot - the total number of nonbase functions for this database
- dertot - the total number of derived functions for this database.

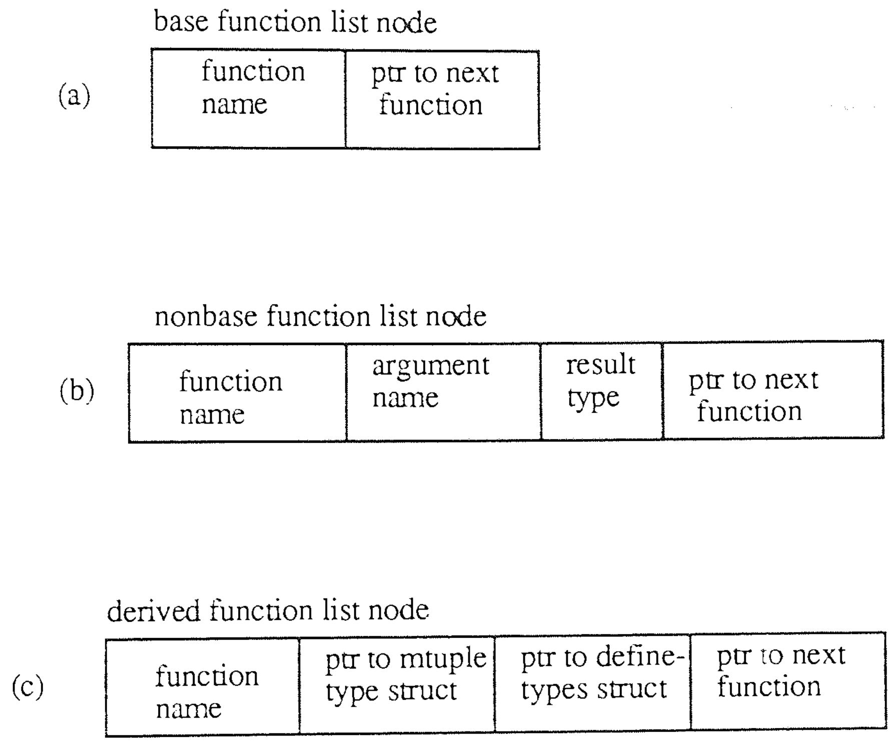


Figure 4.13: Node structures of base, nonbase and derived function lists

At the end of the FQLFE session, the information in these linked list tables and counters is written to the file `<dbname>.func`. First the totals of the base, nonbase, and derived functions are output, in that order. These are then followed by the base, nonbase, and derived functions, again in the same order.

When a database session is invoked, the file `<dbname>.func` is opened, if it exists, and the information is loaded into main memory. The various totals are read in, and using these, the three tables are created. For the derived functions, each function's syntax trees will need to be recreated from the stored information, using the standard C library routine *malloc*. This is because the syntax trees are pointer based, rather than array based.

During language recognition the following information is stored. For example when a statement of the form:

```
DECLARE student() ->> ENTITY
```

is recognised, any necessary checking of semantics and correctness in terms of the relational database is carried out. Then the name of the declared function is entered into the base function table. This is achieved by using the C library routine *malloc* to allocate memory to hold this information. The address of any new base function added is put into the `nextfunc` field of the previous entry. The `nextfunc` field of the current entry is set to `null`. When a declaration such as:

```
DECLARE name(student)->STRING
```

is made, then the function is added to the nonbase function table. Again, when any necessary checking has been done, memory of the relevant size is allocated to contain the name of the function, its argument, and its return-type. The pointers are set as for the above example. When a derived function is specified by a statement such as:

```
DEFINE dept(student) -> department SUCH THAT
       deptno(department) = deptno (student)
```

memory of the relevant size is allocated, and the function name is stored. Memory is then allocated for the arguments of the function. Arguments to defined functions can be complex in their structure and can be highly variable. The mechanism for storing the information about derived functions is simplified in this implementation by storing the complete sub-tree of the argument (this corresponds to `mtuple` according to the Yacc specification of a derived function). Similarly, the return-type of a derived function can also be an arbitrarily complex expression. Again, an effective means to overcome this problem was to store a pointer to the syntax tree of the return-type which, according to the Yacc specification for a derived function, always corresponds to `definetypes`. In this way, detailed information about derived functions is stored efficiently and consistently. The entry in the derived function table will therefore be a node of a simple structure, where the argument and return-types are pointers to the relevant syntax trees. This information will be used when calls to the derived function are made. Calls to this function result in substitutions in this parse tree which is then executed.

If an error is found at any time during language recognition, then no entry is made for that function in any of the tables. At the end of a session, the information contained in the three tables and their counters is written to a file in a predefined format. This is then read back into main memory, when another session of this database takes place.

The `output` module consists of routines, which once again are based on the syntax tree of Daplex statements. This will traverse the syntax tree and will write output to the relevant file:

- whenever it comes across a statement where it checks the value in the `type` field of a structure. It will write out the value of the `type` field.
- whenever it comes across an operator, it will write out the operator
- whenever it comes across an identifier, it will write out the identifier name
- whenever it comes across a string, integer, or constant, it will write out the relevant value.

At the end of a database session, the various tables are resident in main memory and will be lost unless they are saved to file. The base and nonbase function tables are relatively simple to save. They are list structures for which each node of the list is identical. The list is processed and, for each function, its name, argument and return-type are written out to the file as character strings.

The procedure for the derived functions is more complex. As previously stated, derived functions can have arguments and return-types which are arbitrarily long. A simple but effective means of overcoming this problem was found. This was to use the output module. For instance, to write out the mtuple syntax tree for a particular function, the routine output_mtuple is called with the pointer to the mtuple_type structure passed to it as a parameter. Again, to write out the syntax tree for the return-type of the function, the routine output_definetypes is called, with the pointer to the definetypes_type structure passed to it as a parameter.

In a similar way, when the existing functional definition (if any) is read in at the start of a database session, the reading in and creation of the base and nonbase function tables is relatively simple. However, for derived functions, the mtuple and definetypes syntax trees will have to be created from the information contained in the file. The reader routines input_mtuple and input_definetypes are called to do this, and instead of having a pointer passed to these routines, they return pointers, which are then stored in the derived function table as values for the argname and return_type fields. The input routines operate by reading the value into the expected type field, and then, depending on the value of this type field, it expects subsequent values to be of a particular type.

4.11 SECURITY AND INTEGRITY

Ultimately, all responsibility for security is with the underlying DBMS Ingres. This will only allow database access as specified by the underlying relational schema. A possible problem exists in terms of the integrity of the data as it actually is, and as seen through the functional view. A view may be specified for an existing relational database which is validated by FQLFE and is accepted as correct, because it is consistent with the underlying relational schema at the time it is specified, but which is later invalid. This might occur if, at some later point the relational schema is altered or an attribute of a relation, or a complete relation is deleted. When a user of that functional view next attempts to use this database through the functional specification of the database, this functional view may now be incorrect in terms of the underlying relational database if it includes references to any part of the relational schema which no longer exists or has been altered. In such a case, the user statements would be validated by FQLFE against the now inconsistent functional specification and if correct would produce target Quel statements which when executed against the relational database would produce error messages. This is obviously unacceptable. In order to overcome this problem, at system invocation, FQLFE accesses the relational information and the functional information and checks for any inconsistencies. If any are found, these are reported to the user who must then ensure that the view is modified in order to be consistent with the actual information in the database.

4.12 ERROR HANDLING

An error, such as an incorrect database name being specified, is detected and reported to the user immediately. It is not possible to recover from such an error, and therefore the system terminates.

The `monitor` expects its commands to be of a fixed form, that is a '\' followed by a command. Any command that begins with '\' is assumed to be a monitor command. The command is checked against the list of possible commands. If it is a valid command, the relevant action takes place. Alternatively, if the command is not a valid FQLFE monitor command an error message is output on the screen and FQLFE awaits further input from the user. Invalid monitor commands do not cause system termination. Any user input which is not preceeded by '\' is assumed to be a Daplex statement by the `monitor`, hence there is no error recovery mechanism necessary for this phase. Instead the user's input is written to the workspace file, regardless of whether it is valid Daplex input. The contents of the workspace file are verified at a later stage.

During lexical analysis, any input that does not conform to the specified lexical patterns results in a special token. This token is passed to the syntax analyser in the same way as other tokens. Again, this means that there is no need for an error recovery strategy here.

During syntax analysis, any unexpected input generates an error message, After locating the error, an attempt is made to continue the syntax analysis of subsequent statements.

Errors discovered during semantic analysis are again reported when found, and the remainder of the statement is ignored. Semantic analysis is continued with subsequent statements.

Unix system errors, such as read or write errors, or inability to allocate main memory or spawn child processes, are all detected and reported by FQLFE but there is no attempt to recover from these as it would not be meaningful (nor feasible).

4.13 LIMITATIONS

This implementation of FQLFE does not include the facility to update database information. This is not due to any technical problems, but due to time restrictions. Data retrieval has been given priority, since this is the most fundamental part of data manipulation. Even in terms of updates, the most important factor is to be able to select the data for updating. By implementing database querying within FQLFE, its data selection abilities have been clearly demonstrated. To incorporate updates, additions need to be made to the Lex and Yacc specifications and the relevant transformation routines need to be added to generate the update statements in the target Quel language. This should not pose any problems as the FQLFE system provides a structured framework to be able to do this without difficulty.

Specifying Constraints and Triggers is another area which has not been implemented in FQLFE due to time restrictions, however discussion of how this could be done is necessary. Constraint specifications can involve an instruction to abort any update transaction which causes failure of a given pre-defined condition. As an example of this, consider the constraint that a department's head must belong to the same department.

```
DEFINE CONSTRAINT nativemgr(department) ->

     dept(manager(department)) = department
```

Though this type of constraint is specifiable in Daplex, there is no obvious equivalent Quel statement that it can be transformed into. Therefore this type of Daplex statement has not been made available in FQLFE. This could be provided by maintaining information in FQLFE system tables which would need to be accessed to verify that a particular update did not violate the constraint. However, Quel allows the specification of simple integrity constraints such as: 'ensure that all employees have positive salaries'. This would be expressed in Quel as:

```
range of e is employee

define integrity on e is e.salary > 0
```

In Daplex this can be expressed in two ways:

- As a TRIGGER statement which causes the name of the employee to be printed when this constraint is violated.

```
DEFINE TRIGGER illegalsal(employee) ->

      salary(employee) <= 0

PRINT name(employee)
```

- As a CONSTRAINT statement which abandons any update which would cause the condition to be violated.

```
DEFINE CONSTRAINT possal(employee) - >
          salary(employee) > 0
```

Though there appear to be limitations of the underlying DBMS in terms of the facilities for the specification of constraints, these could be overcome in the manner suggested above. Further investigation is required to consider this fully.

4.14 AUTOMATIC FUNCTION GENERATION

Use of FQLFE involves a definition phase, whereby the relations in the underlying relational database are specified in terms of the Daplex data model using the Daplex query language. During this research, a system was developed which could do this automatically. The system `autogen` uses the catalogue information about the relational database to do this. Relations in the database are assumed to correspond to base functions, while attributes of those relations correspond to nonbase functions. Derived functions could by this definition, correspond to views. To execute the system, the following command must be specified:

```
autogen <dbname>
```

where `<dbname>` specifies an existing Ingres database for which the user wishes to generate the definition of the relational view. When the system is invoked, it verifies that `<dbname>` has been specified. It then generates a child process in which it executes the Ingres DBMS. It then selects the relevant information and enters it into a table of the structure specified in figure 4.2. It then processes this table. For each relation in the database, it produces a statement:

```
DECLARE <relname> ( )->ENTITY
```

where `<relname>` specifies the relevant relation. For each attribute, it produces a statement:

```
DECLARE <attrname> ( <relname> )-> <rettype>
```

where `<attrname>` specifies the relevant attribute of the particular relation, `<relname>`, and `<rettype>` specifies the return-type of the function.

As these Daplex statements are produced, they are written to a file. The FQLFE system is then invoked with this file as input and processes these definitions verifying them as relevant. At the end, it saves the function definitions to the file `<dbname>.func`, which will be loaded up for that users next database session on FQLFE, involving that database.

This concept of automatic generation could be enhanced by attempting to produce the Daplex functions specifying the functional view automatically, in the same way. However this would need further investigation, and will almost certainly involve interaction with the user, since it requires knowledge about the data relationships.

4.15 CONCLUSIONS

This chapter has presented a prototype implementation of a functional query language front-end to a relational database system. This is useful as an alternative view through which the database can be accessed. In addition, it provides an example of how a functional query language could be used as a front-end to an existing database system. This Daplex front-end could be adapted to provide an interface to a network of dissimilar database management systems. This would involve using existing Daplex descriptions for each of the local databases in the network. These in turn would be accessible via view mechanisms as a common unified view of the entire network database. Daplex provides the global language by which the database could be accessed.

The FQLFE system provides a facility for additional semantic information to be added to the database. The user does not have to be aware of the explicit linking across relations that would otherwise be necessary. These links are represented in the functional schema of the relational database. The front-end uses this information when queries are made to the database and generates the relevant target language code.

A system such as this could be adapted to front-end a different DBMS than the one used in this research. It could also be adapted to generate a different target language than the one used.

CHAPTER 5

FQLFE SYSTEM USAGE

5.1 TEST DATABASE

The database application used to test the FQLFE system is based on a database whose schema is specified in figure 5.1 below. The database is based on an example described by Lacroix & Pirotte (1978). This example was used as a basis for the testing of FQLFE since it (or variations on it) are often used as an example application. Lacroix & Pirotte (1978) also present a list of queries that they have maintained during their study and design of query languages over a number of years. The testing of FQLFE involved using a selection of queries from this paper.

To use this database as a Daplex database requires that this relational description is expressed in terms of Daplex functions. The information about the relations in the database was used to develop the Daplex description for the database. The relevant declarations are specified in figure 5.2.

```
employee(empno, name, salary,  managerno, deptno)

sales(deptno, itemno, vol)

supply(compno,deptno, itemno, vol)

supplier(compno, name, address)

department(deptno, name, floor)

item(itemno, name, type)
```

Figure 5.1: The Relational Description

```
DECLARE employee( ) ->> ENTITY
DECLARE empno( employee ) -> INTEGER
DECLARE salary( employee ) ->INTEGER
DECLARE managerno( employee ) ->INTEGER
DECLARE deptno( employee ) -> INTEGER
DECLARE name( employee ) -> STRING

DECLARE sales( ) ->> ENTITY
DECLARE deptno( sales ) -> INTEGER
DECLARE itemno( sales ) -> INTEGER
DECLARE vol( sales ) -> INTEGER

DECLARE supply( ) ->> ENTITY
DECLARE compno( supply ) -> INTEGER
DECLARE deptno( supply) -> INTEGER
DECLARE itemno( supply) -> INTEGER
DECLARE vol( supply ) -> INTEGER

DECLARE supplier( ) ->> ENTITY
DECLARE compno( supplier ) -> INTEGER
DECLARE address( supplier ) -> STRING
DECLARE name( supplier ) ->STRING

DECLARE  department( ) ->> ENTITY
DECLARE deptno( department ) -> INTEGER
DECLARE floor( department ) -> INTEGER
DECLARE name( department ) -> INTEGER

DECLARE item( ) ->> ENTITY
DECLARE itemno( item ) -> INTEGER
DECLARE type( item ) -> INTEGER
DECLARE name( item ) -> STRING
```

Figure 5.2: Daplex Description of Relational Schema

```
DEFINE  dept( employee ) ->> department SUCH THAT

      deptno( employee ) = deptno( department )

DEFINE floor( sales) ->> department SUCH THAT

      deptno( sales ) = deptno( department )

DEFINE sold( item ) ->> sales SUCH THAT

      FOR SOME department

            itemno( sales ) = itemno( item ) AND

            deptno( department ) = deptno( sales )

DEFINE deptsells(employee) ->> sales SUCH THAT

      deptno( sales) = deptno( employee)

DEFINE suppitem(supply)->> item SUCH THAT

      itemno(supply) = itemno(item) AND

      compno(supply) = compno(supplier)

DEFINE  supplied(item) ->> supply SUCH THAT

      itemno(item) = itemno(supply)

DEFINE supplies(item) ->> supplier SUCH THAT

      FOR SOME supply

            itemno(supply) = itemno(item) AND

            compno(supplier) = compno(supply)

DEFINE comp(supply) ->> supplier SUCH THAT

      compno(supplier) = compno(supply)

DEFINE itemsold(department)->> item SUCH THAT

      FOR SOME sales

            itemno( sales ) = itemno( item ) AND

            deptno( department ) = deptno( sales )
```

Figure 5.3: Daplex Description for Functional View

In order to use Daplex query statements against this database, a number of definitions are necessary to specify the functional view of the relational database (the reasons for this are given in section 4.1). The Daplex statements developed to specify this functional view are given in figure 5.3. Lacroix & Pirotte (1978) also specifies example queries in a number of relational query languages, including Quel. This was incorporated into the testing, by using the Quel expressed queries to verify the correct transformation of the Daplex expressed queries.

5.2 DATABASE QUERYING

This section gives examples of queries made against the database. These are specified in the following format. First the query is given in ordinary English, then the Daplex representation of the query is given. Then the query is given in its transformed Quel state. Any relevant features of either the Daplex or the Quel query are indicated.

5.2.1 Query 1.

Find the names of employees in the toy department.

Daplex

```
FOR EACH employee SUCH THAT
      name(dept(employee)) = "TOY"
PRINT name(employee)
```

Quel

```
range of e is employee
range of d is department
retrieve (e.name)
where
```

```
        e.deptno = d.deptno and
        d.name = "TOY"
```

This query is an example of using a Daplex derived function - `dept`. The FQLFE system transforms this by substituting the relevant base functions into the query syntax tree. In addition the query shows the use of nested function calls, with functions calling derived functions mixed with those calling base functions.

5.2.2 Query 2

Find the items sold by departments on the second floor.

Daplex

```
FOR EACH item SUCH THAT
        FOR SOME sold(item)
                floor(department) = 2
PRINT name(item), name(department)
```

Quel

```
range of i is item
range of s is sales
range of d is department
retrieve (i.name, d.name) where
        s.itemno = i.itemno and
        d.deptno = s.deptno and
        d.floor = 2
```

This query is an example of universal quantification. It is a multi-table join involving three tables in the underlying database, these are `item`, `sales`, and `department`. In the Daplex version of the query, this is hidden from the user. The FQLFE system uses the previously defined derived function to transform it into the equivalent Quel query.

5.2.3 Query 3

Find the salary of Anderson's manager.

Daplex

```
FOR EACH e IN employee SUCH THAT
      FOR  SOME e1 in employee SUCH THAT
            name(e1) = "ANDERSON" AND
            managerno(e1) = employeeno(e)
PRINT salary(e)
```

Quel

```
range of e is employee
range of e1 is employee
retrieve (e.salary) where
      e1.name = "ANDERSON" and
      e1.managerno = e.employeeno
```

An alternative way of expressing this query in Daplex is:

```
FOR EACH employee SUCH THAT
      name(employee) = "ANDERSON"
PRINT salary(manager(employee))
```

In this query the symbol `employee` is used in two distinct senses. In the first line, it refers to the set of `employee` entities, while in the other lines, it is a looping variable. It is bound successively to the members of the iteration set `employee` implicitly. The Quel query can only be expressed by multiple range variables ranging over the same relation. Daplex provides the means to explicitly specify multiple variables ranging over the same set. The first Daplex formulation of the query in this section makes use of this feature.

5.2.4 Query 4

Find the names of the employees who earn more than their managers.

Daplex

```
FOR EACH e IN employee SUCH THAT
      FOR SOME e1 in employee
            salary(e) > salary(e1) AND
            managerno(e) = empno(e1)
PRINT name(e)
```

Quel

```
range of e is employee
range of e1 is employee
retrieve (e.name) where
      e.managerno = e1.empno and
      e.salary > e1.salary
```

In Quel, this query can only be expressed with two range variables ranging over the same set.The above representation of the query makes use of explicit looping variables, `e`, and `e1`.

5.2.5 Query 5

Find the names of the employees whose salary equals that of their manager.

Daplex

```
FOR EACH e in employee SUCH THAT
        FOR SOME e1 in employee
                managerno(e) = empno(e1) AND
                salary(e) = salary(e1)
PRINT name(e)
```

Quel

```
range of e is employee
range of e1 is employee
retrieve (e.name) where
        e.managerno = e1.empno and
        e.salary = e1.salary
```

This query makes use of the relational operator `equals`.

5.2.6 Query 6

Find the names of the employees who are in the same department as their manager (as an employee).

Daplex

```
FOR EACH e in employee SUCH THAT
        FOR SOME e1 in employee
                managerno(e1) = empno(e)  AND
                deptno(e) = deptno(e1)
PRINT name(e)
```

Quel

```
range of e is employee
range of e1 is employee
retrieve (e.name) where
      e.deptno = e1.deptno AND
      e.empno = e1.managerno
```

Again the query involves explicit looping variables, to range over the same set. In the transformed query the two range variables range over the same relation.

5.2.7 Query 7

List the departments having an average salary greater than £25000.

Daplex

```
FOR EACH employee SUCH THAT
      FOR SOME dept(employee)
            AVERAGE (salary(employee)
                  OVER deptno(employee))
                  > 25000
PRINT name(department)
```

Quel

```
range of e is employee
range of d is department
retrieve (d.name) where
      d.deptno = e.deptno AND
      avg(e.salary by e.deptno) > 25000
```

This function expressed in Daplex has a number of features. It involves the use of a derived function `dept`. It provides an example of using an aggregate function, AVERAGE. In addition, the OVER operator is used to ensure that every resulting value is included in the average, regardless of whether there are any duplicates present. The query expressed in Quel involves a join across two tables, `employee` and `department`. In addition, it involves use of the Quel aggregate function `avg`.

5.2.8 Query 8

List the name and salary of the managers who manage more than 10 employees.

Daplex

```
FOR EACH e in employee SUCH THAT
      FOR SOME el in employee
            managerno(el) = manager(e) AND
            count(manager(el) OVER el) > 10
PRINT name(e), salary(e)
```

Quel

```
range of e is employee
range of el is employee
retrieve (e.name, e.salary) where
      e.empno = el.managerno AND
      count(el.empno by el.managerno) > 10
```

The interesting features of this query are that it involves nested functions both in the body of the FOR statement and also in the PRINT statement. It also provides an example of a Daplex query using the aggregate function COUNT. The argument to the aggregate function involves the OVER operator. In addition, the query involves use of both logical and relational operators. The equivalent Quel query involves use of two range variables ranging over the same table. Similarly, it involves the Quel aggregate function `count`, with an argument which involves use of the `BY` clause.

5.2.9 Query 9

List the names of the employees in the shoe department who have a salary greater than £25000 together with the names of their managers.

Daplex

```
FOR EACH e IN employee SUCH THAT
      FOR SOME el in employee
            managerno(e) = empno(el) AND
            name(department) = "SHOE" AND
            deptno(e) = deptno(department) AND
            salary(e) > 25000
PRINT name(e), name(el)
```

Quel

```
range of e is employee
range of el is employee
range of d is department
retrieve (e.name, el.name) where
e.managerno = el.empno AND
e.deptno = d.deptno AND
d.name = "SHOE" AND
e.salary > 25000
```

This query also involves two range variables ranging over the same set. However, it also involves a range variable ranging over a different set. It specifies the two range variables ranging over the set employee explicitly. The Quel query involves three range variables with two of those range variables ranging over the same table. In addition the query involves a join across two tables, department and employee. It also shows the use of multiple logical AND operators, in addition to the relational greater than operator.

5.2.10 Query 10

List the names of the employees who make more than any employees in the shoe department.

Daplex

```
FOR EACH  employee SUCH THAT
      salary(employee) >
      MAXIMUM(salary(employee) SUCH THAT
            name(dept(employee)) = "SHOE")
PRINT name(employee)
```

Quel

```
range of e is employee
range of d is department
retrieve (e.name) where
      e.salary > max(e.salary where
      e.deptno = d.deptno
      AND d.name = "SHOE")
```

This query involves two range variables ranging over different sets. Also the right hand side of the logical operator greater than is an aggregate function call, MAXIMUM. In addition, it is an example of an aggregate function call which involves a SUCH THAT clause in the argument to the function call. The target Quel query is an example of using the aggregate function `max`, with the argument to the function involving a `where` clause, and also a logical `and` operator in the argument to the function call.

5.2.11 Query 11

Among all departments with total salary greater than £40000, find the departments which sell paperbacks.

Daplex

```
FOR EACH department SUCH THAT
      FOR SOME employee
            TOTAL (salary(employee)
                  OVER deptno(sales) SUCH THAT
                        deptsells(employee))
                        > 100000
            AND name(itemsold(department))
                              = "PAPERBACK"
PRINT name(department)
```

Quel

```
range of s is sales
range of e is employee
range of d is department
range of i is item
retrieve (d.name) where
sum(e.salary by s.deptno where
s.deptno = e.deptno)
```

```
> 100000 and
i.name = "PAPERBACK" and
i.itemno = s.itemno and
s.deptno = d.deptno
```

The Daplex version of this query uses nested `FOR SOME` statements. It involves calls to two derived functions, `sold` and `dept`, in the same query. It also provides an example of a query involving the aggregate function `TOTAL`. The Quel transformation of this query is an example of a multi-table join, involving the relations `sales`, `employee`, and `department`. It is also an example of a Quel query involving use of the `sum` aggregate function, with the function call involving a `by` operator.

5.2.12 Query 12

Find the companies that supply pens.

Daplex

```
FOR EACH item SUCH THAT
      FOR SOME supplies(item)
            name(item) = "PEN"
PRINT name(supplier)
```

Quel

```
range of s is supply
range of s1 is supplier
range of i is item
retrieve (s1.name) where
      i.name = "PEN" and
      i.itemno = s.itemno and
```

```
s.compno = sl.compno
```

This query also uses a derived function for which FQLFE substitutes the corresponding syntax tree during transformation to arrive at the target Quel query shown above. The Quel query has two range variables which range over sets whose first letter is the same. FQLFE automatically deals with this, and correctly allocates different range variables to range over these tables.

5.2.13 Query 13

Find the companies that supply an item other than pens.

Daplex

```
FOR EACH item SUCH THAT
      FOR SOME supplies(item)
            name(item) NE "PEN"
PRINT name(supplier)
```

Quel

```
range of s is supply
range of sl is supplier
range of i is item
retrieve (sl.name) where
      i.name != "PEN" and
      i.itemno = s.itemno and
      s.compno = sl.compno
```

Query 5.2.13 expressed in both Daplex and Quel is an example of a query involving use of the relational operator `not equals`. The Quel query involves a multi-table join.

5.2.14 Query 14

List the items supplied by exactly one supplier.

Daplex

```
FOR EACH item SUCH THAT
        FOR SOME supplied(item)
                COUNT (compno(supply)
                        OVER itemno(supply)) = 1
PRINT name(item)
```

Quel

```
range of s is supply
range of i is item
retrieve (i.name) where
        i.itemno = s.itemno and
        count(s.compno by s.itemno)  = 1
```

This query expressed in Daplex is an example involving an aggregate function, COUNT with the argument to the aggregate function involving both an OVER operator and a SUCH THAT clause. The target Quel query involves the aggregate function, count and its argument involves both a by clause and a where clause.

5.2.15 Query 15

List the companies that are the only supplier of some item.

Daplex

```
FOR EACH supply SUCH THAT
        FOR SOME comp(supply)
```

```
                COUNT (compno(supply)
                      BY itemno(supply)) = 1
        PRINT name(supplier)
```

Quel

```
        range of s is supply
        range of s1 is supplier
        retrieve (s1.name) where
              s1.compno = s.compno and
              count(s.compno by s.itemno) = 1
```

This is an example of a query involving an aggregate function call in the body of a `FOR SOME` statement. In addition it provides an example of a query where the value returned by the aggregate is of importance, that is, only those entities which evaluate to a count of one are selected. The Quel statement which results involves an aggregate function call in the `where` clause. Again the value returned by the aggregate function call `count` is compared to one.

5.2.16 Query 16

List the companies that are the suppliers of at least 3 items.

Daplex

```
        FOR EACH supply SUCH THAT
              FOR SOME comp(supply)
                    COUNT (itemno(supply)
                    OVER compno(supply)
                    SUCH THAT
                          (COUNT (compno(supply)
                          OVER itemno(supply))
```

```
                  = 1)

            >= 3

    PRINT name(supplier)
```

Quel

```
    range of s is supply

    range of s1 is supplier

    range of i is item

    retrieve (s1.name) where

          s1.compno = s.compno and

          count(s.itemno by s.compno where

          count(s.compno by s.itemno) = 1 )

                >=3
```

Query 16 has a number of features, which include the use of derived functions and a call of an aggregate function whose argument consists of a call to another aggregate function (that is nested aggregate function calls). In addition, the outer aggregate call consists of both an `OVER` clause, and a `SUCH THAT` clause. The target Quel query also involves nesting of the aggregate function `count`. Again, the example involves a `where` clause in the aggregate function call argument.

5.2.17 Query 17

For each item give its type, the departments which sell the item and the floor of these departments.

Daplex

```
    FOR EACH item SUCH THAT

          sold(item)

    PRINT name(item), type(item), name(department),
        floor(department)
```

Quel

```
range of i is item

range of s is sales

range of d is department

retrieve (i.name, i.type, d.name, d.floor) where

        i.itemno = s.itemno and

        s.deptno = d.deptno
```

Query 17 is an example involving both derived functions and a command to print values from more than one set. The Quel query involves existential quantification and is a multi-table join, with the items in the target list being selected from two of those tables.

5.2.18 Query 18

Find the average salary of the employees in the shoe department.

Daplex

```
FOR EACH employee

PRINT AVERAGE (salary(employee)

        SUCH THAT name(dept(employee)) = "SHOE")
```

Quel

```
range of e is employee

range of d is department

retrieve (a = avg(e.salary where

        d.name = "SHOE" and

        d.deptno = e.deptno))
```

The interesting aspect of query 18 is that the Daplex expression of the query involves an aggregate function call in the target list specified by the `PRINT` command. Similarly, the Quel query involves an aggregate function call in the target list specified by the `retrieve` clause.

5.2.19 Query 19

Give, for each department, the average salary of the employees.

Daplex

```
FOR EACH employee SUCH THAT
      FOR SOME dept(employee)
PRINT name(department),
      AVERAGE(salary(employee)
      OVER deptno(employee) SUCH THAT
            dept(employee))
```

Quel

```
range of e is employee
range of d is department
retrieve (d.name, a = avg(e.salary
      by e.deptno)) where
      e.deptno = d.deptno
```

Query 19 exhibits features similar to those of query 18.

5.2.20 Query 20

Give, for each department, its floor and the average salary.

Daplex

```
FOR EACH employee SUCH THAT
      dept(employee)
PRINT name(department), floor(department),
      AVERAGE (salary(employee)
            OVER deptno(employee))
```

Quel

```
range of e is employee
range of d is department
retrieve (d.name, d.floor,
      a = avg(e.salary by e.deptno))
      where e.deptno = d.deptno
```

This is another example of a query which involves an aggregate function call in the target list which, in the Daplex version, involves an `OVER` clause and, in the Quel version, involves a `by` and `where` clause.

5.2.21 Query 21

List companies that supply a total volume of items of types A and B which is altogether greater than 1000.

Daplex

```
FOR EACH supply SUCH THAT
      FOR SOME item SUCH THAT
```

```
                    FOR SOME comp(supply)

                            TOTAL (vol(supply)

                            OVER compno(supply)

                            SUCH THAT supplied(item)

                                    AND type(item) = "A")

                            +

                            TOTAL (vol(supply)

                            OVER compno(supply)

                            SUCH THAT supplied(item)

                                    AND type(item) = "B")

                            > 1000

        PRINT name(supplier)
```

Quel

```
        range of s is supply

        range of i is item

        range of s1 is supplier

        retrieve (s1.name) where

                s1.compno = s.compno and

                sum(s.vol by s.compno

                where s.itemno = i.itemno

                        and i.type = "A")

                +

                sum(s.vol by s.compno

                        where s.itemno = i.itemno

                                and i.type = "B")

                > 1000
```

This is an example of a relatively complex query both in Daplex and Quel. It involves use of the mathematical operator '+' and also complex aggregate function calls.

5.2.22 Query 22

List the employees in the shoe department and the difference of their salaries with the average salary of the department.

Daplex

```
FOR EACH employee SUCH THAT
      FOR SOME dept(employee)
            name(department) = "SHOE"
PRINT name(employee,
      salary(employee) -
            AVERAGE (salary(employee)
                  OVER deptno(employee) SUCH THAT
                        name(dept(employee)) = "SHOE")
```

Quel

```
range of e is employee
range of d is department
retrieve (e.name,
      a = e.salary - avg(e.salary by e.deptno where
            e.deptno = d.deptno and
            d.name = "SHOE")) where
                  e.deptno = d.deptno
                  and d.name = "SHOE"
```

This is an example of using the mathematical operator '-' in the target list specification in both the Daplex query and the Quel query.

5.2.23 Query 23

List the employees in the shoe department and the difference of their salaries with the average salary computed for all departments.

Daplex

```
FOR EACH employee SUCH THAT
      FOR SOME dept(employee) SUCH THAT
            name(department) = "SHOE"
PRINT name(employee),
      salary(employee) - AVERAGE (salary(employee))
```

Quel

```
range of e is employee
range of d is department
retrieve (e.name, a = e.salary - avg(e.salary)) where
      d.name = "SHOE" and d.deptno = e.deptno
```

This is an example of a query which, as well as involving the mathematical subtraction operator, involves a `where` clause in the target list, which is not in the aggregate function call.

5.2.24 Query 24

List each employee and the difference of his salary and the average salary of the department where he works.

Daplex

```
FOR EACH employee

PRINT name(employee),

      salary(employee) - AVERAGE (salary(employee)

            OVER deptno(employee))
```

Quel

```
range of e is employee

retrieve (e.name,

      a = e.salary - avg(e.salary by e.deptno))
```

This is another example of a query involving the mathematical subtraction operator, as well as an aggregate function call.

5.2.25 Query 25

What is, for each supplier, the average number of items per department that the supplier supplies?

Daplex

```
FOR EACH supply SUCH THAT

      comp(supply)

PRINT name(supplier),

      AVERAGE (COUNT (itemno(supply)

            OVER deptno(supply),compno(supply))

                  OVER compno(supply))
```

Quel

```
range of s is supply

range of s1 is supplier
```

```
                                              S
retrieve (s1.name, a = avg(count(s1.itemno
      by s.deptno, s.compno)
            by s.compno)) where
                  s1.compno = s.compno
```

This involves two different aggregate function calls with one as the argument to the other.

5.2.26 Query 26

For each department find the average salary of the employees who earn more than the average salary of the department.

Daplex

```
FOR EACH employee SUCH THAT
      dept(employee)
PRINT name(department), AVERAGE (salary(employee)
      OVER deptno(employee) SUCH THAT
            salary(employee) >
                  AVERAGE (salary(employee)
                        OVER deptno(employee)))
```

Quel

```
range of e is employee
range of d is department
retrieve (d.name, a = avg(e.salary
      by e.deptno where
            e.salary > avg(e.salary by e.deptno)))
            and d.deptno = e.deptno
```

This is a relatively complex query, which in the Daplex version has a target list involving nested aggregate function calls, where one of the aggregate function calls is an operand for the logical operator `greater than`.

5.2.27 Query 27

Give the overall average of the average salary per department.

Daplex

```
FOR EACH employee
PRINT AVERAGE (AVERAGE (salary(employee)
      OVER deptno(employee)))
```

Quel

```
range of e is employee
retrieve (a = avg(avg(e.salary by e.deptno)))
```

This is another example of a nested aggregate function call.

5.2.28 Query 28

List, for each employee, his salary, the average salary of the department where he works and the difference of his salary and the average salary of the department where he works.

Daplex

```
FOR EACH employee
PRINT name(employee),
      AVERAGE (salary(employee)
            OVER deptno(employee)),
                    salary(employee) -
                         AVERAGE (salary(employee)
                              OVER deptno(employee))
```

Quel

```
range of e is employee
retrieve (e.name,
     a = avg(e.salary by e.deptno)
          b = e.salary - avg(e.salary by e.deptno))
```

Another example involving complex use of aggregate functions.

5.2.29 Query 29

Find the companies that do not supply pens.

Daplex

```
FOR EACH supply SUCH THAT
     FOR SOME supplier
          COUNT(compno(supply)
               OVER compno(supplier)
                    SUCH THAT name(suppitem
                              (supply)) = "PEN")
= 0
PRINT name(supplier)
```

Quel

```
          range of i is item
range of s is supply
range of s1 is supplier
```

```
retrieve (s1.name) where
      count(s.compno by s1.compno where
            s1.compno = s.compno and
            s.itemno = i.itemno and
            i.name = "PEN" ) = 0
```

This is an example of the use of the Quel aggregate function `count` and using it to specify negation in the format: `count(........) = 0.`

5.2.30 Query 30

Find the items sold by no department on the second floor.

Daplex

```
FOR EACH item SUCH THAT
      FOR SOME sold(item)
            COUNT (deptno(sales)
                  OVER itemno(sales) SUCH THAT
                        floor(floor(department) = 2) =
0
PRINT name(item)
```

Quel

```
range of d is department
range of s is sales
range of i is item
retrieve (i.name) where
      d.deptno = s.deptno and
      i.itemno = s.itemno and
      count(s.deptno by s.itemno where
                    s .deptno = d.deptno and
                          d.floor = 2) = 0
```

This is an example of a query for which the set whose entities are being counted by the aggregate function COUNT, should be empty.

5.3 CONCLUSIONS

In this chapter the use of the FQLFE system has been demonstrated by means of an example implementation. In order to use the FQLFE system, the schema of the relational database as it exists on the Ingres DBMS system has to be specified in terms of Daplex. The definitions necessary to specify this as a functional database under the FQLFE system have been given. Further, Daplex definitions are necessary to establish a particular user's view of the database. The use of this user's view of the database has been demonstrated by examining the sort of queries that can be made against the underlying relational database using the functional query language. The implementation of the test database and queries against it have verified the possibility that a database can be queried in terms of a query language other than the ones which are specifically for the particular data model upon which it is based. In terms of querying a database, such a system certainly enhances the database environment. However in terms of updating, it would cause problems, as there is now an increased possibility of inconsistencies between a user's view of the database and the underlying database. This possibility can of course be minimised if particular users consistently use the same interface for accessing the database irrespective, of which of the interfaces they use.

CHAPTER 6

CONCLUSIONS

Query languages for particular data models represent the primitive operations available under that data model. The language interface is an important part of the DBMS environment. Interfaces to databases consist of a variety of approaches. These include graphical interfaces and form-based interfaces as well as query language interfaces. Some modern DBMS provide more than one query language interface but only where the query languages are closely related and based on the same data model.

This research attempts to take this one step further, to provide an additional query language interface in the DBMS environment, whose underlying data model is different to that of the DBMS to which it is interfaced. There is a number of issues which determine the characteristics of the selected query language. The most important of these is the data model upon which it is based. Query languages based on particular data models address the data structures of the data model and the semantics of the expressions in the language can be expressed with operations of the data model. In general, effective use of the database depends to a large extent upon the user understanding of the database view. By providing a system such as FQLFE the user has a choice as to the database model. By presenting an interface based on a functional data model to a relational DBMS, the user's view can now be based around functions rather than relations. This also has the advantage in that because large amounts of time and money have been invested by organisations in their DBMS and to change completely from their existing DBMS to a DBMS based on a different data model is unlikely to be practical. In addition, from a user's point of view, it is better that a choice is available than to have a particular situation imposed.

The suitability of particular query languages to be the target or front-end language were examined. It was noticed that the features of the query language are determined by the data model upon which the query language is based. The data structures available and the operations allowed on them are features of the underlying data model. In selecting the interface and target languages a number of characteristics are important .

The data model of the target DBMS should be one that is widely available. Since relational database technology is widespread, this is the obvious choice for the target DBMS. In addition, relational DBMS have a formal theoretical basis and behave in well-defined ways. The number of concepts upon which it is based are relatively small. The Ingres DBMS is widely available and it runs in many different environments (mainly Unix). This is advantageous, since Unix systems are widespread and their use is continually increasing. In addition, the fact that the FQLFE system is written in C means that it is highly portable. By using a front-end based on a data model such as the functional data model, more semantic information about the database can be expressed and some of the restrictive practices or problems of the underlying data model can be avoided. At the same time, the existing DBMS can continue to be used in the same way as before, if required.

A number of query languages were examined for other features. Amongst these was simplicity. The relational languages are generally simpler than those based on other models, at least in terms of the minimum number of concepts required to get started and the view that simple operations should be expressed simply. In the same way it is important that the target language is relatively simple, otherwise transformation to the target language would become an insurmountable hurdle.

The format of the the target language is also important. For instance, graphical interfaces are not suitable as target languages, because the whole basis of their development is their display feature and many other compromises will have been made to enhance this. They were considered as languages for the front-end system, but were rejected since they were either relational model based (and this had already been selected as a suitable target data model) or they required special equipment, for example, light pens and bit mapped screens. It was felt that the need to use such specialised equipment conflicts with the concept of a simple alternative, readily available. Specialised equipment would mean reduced availability. The language selected, Daplex, is based on keywords. Quel proved suitable as a target language due to its format. It also uses keywords and is well structured.

The FQLFE implementation is self-contained and interacts with the DBMS at only a few points, these points of interaction are well-defined: the underlying DBMS is accessed to provide information about the underlying database structure and to execute the transformed queries. These well-defined points of interaction mean that FQLFE will not interfere with or adversely affect the underlying DBMS. Further, it means that the FQLFE system is highly portable, and can be altered relatively easily for another machine environment.

The FQLFE system development is structured in such a way that it consists of four main components: the user interface, the language recogniser, the language transformer and the database access module. By developing the system in this way its adaptability has been enhanced. The FQLFE system could be adapted to generate a different query language than the one selected. This would require changes to be made to a specific localised part of the FQLFE system. Thus, it presents a framework for other front-ends, possibly on other hardware.

Ingres provides a number of system level commands, that is, commands that operate at the level of the machine's operating system (Unix), rather than at the Ingres DBMS level. Examples of such commands are `createdb` and `destroydb`. It would be possible to provide such commands in the FQLFE system. This possibility was rejected for a number of reasons. Firstly, any of these commands would have to be provided as utilities to be run at the operating system level. They would have to involve calling the existing system level commands to maintain the consistency that was attempted with the FQLFE system in trying to ensure a 'clean' interface between the FQLFE system and the underlying DBMS. In effect, the commands would have exactly the same function as the provided system level commands and there is not any obvious or useful way in which they could be enhanced that would justify the extra level of command that would need to be introduced to implement them. For these reasons, in a situation where a system such as FQLFE was available, it would be likely that the database administrator would be the only user likely to require these commands. In addition, they would be used on fewer occasions than the database query language. Therefore there are no obvious advantages in providing these again within FQLFE.

The FQLFE system is implemented as a front-end to the Ingres DBMS, and as a result suffers from the problem that all its retrieval times are a sum of the time it takes FQLFE to transform the command into target language statements and the time taken by the Ingres DBMS to execute those commands. The FQLFE system can if necessary be refined to minimise the time for it to produce its output, but this is the only factor in the calculation that can be affected. There will always be the issue of the amount of time taken by the underlying DBMS to produce a response to the query. However, this is completely outside the control of the FQLFE system. One factor that could be improved results from the current implementation invoking the Ingres DBMS system every time a query is made to it. This is relatively time consuming. Timings could be improved by invoking the process running the Ingres DBMS once only at the beginning of the session and connecting it to the parent process by the means of a Unix *pipe*. The process running the Ingres DBMS would thus read its input from this pipe rather than from the file as at present.

This research has shown that systems such as FQLFE are feasible, but they do involve large amounts of software and there are other limitations. The features that are provided in FQLFE are a subset of both the front-end language and the target language. This is due to the fact that there are some features of the front-end language for which there may not be any equivalent that is attainable by simple transformations. This need not always be impossible to overcome, for example, more than one action in the target language could result, or an action involving execution of other software could result. Despite this there may be some aspects of the front-end language which are not transformable into the target language. This is expected, to a certain degree, since the languages are based on different models and this diversity is the very reason for selecting them. The full extent of these limitations needs further investigation. FQLFE could play an important role in such an investigation since it provides a suitable framework for further exploration of the true extent of any limitations.

REFERENCES

Astrahan, M.M, and Chamberlin, D.D (1975) *Implementation of a Structured English Query Language*, Communications of the ACM, **18**(10), October, 1975.

Atzeni, P and Chen, P.P (1983) *Completeness of Query Languages for the Entity-Relationship Model*, in Chen (1983).

Bachman, C.W (1973) *The Programmer as Navigator*, Comunications of the ACM, **16**(11).

Bachman, C.W, and Dayal, M (1977) *The Role Concept in Data Models*, Proc. 3rd Int. Conf. On Very Large Databases, October 1977, Japan.

Backus, J (1978) *Can Programming be Liberated from the Von Neuman Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, **21**(8).

BCS (British Computer Society Query Language Group) (1981) Query Languages: A Unified Approach, Heydon and Sons.

Brodie, M.L (1984) *On The Development of Data Models*, in Brodie *et al.* (1984).

Brodie, M.L, Mylopoulos, J and Schmidt, J.W (eds) (1984) Conceptual Modelling, Springer-Verlag, Berlin.

Buneman, P and Frankel, R.E (1979) *FQL- A Functional Query Language*, Proceedings of the ACM-SIGMOD Conference.

Buneman, P, Frankel, R.E and Nikhil, R (1982) *An Implementation Technique for Database Query Languages*, ACM Transactions on Database Systems, 7(2), pp. 164-186.

Chamberlin, D.D (1976) *Relational Data-Base Management Systems*, Computing Surveys, **8**(1).

Chen, P (ed) (1983) Information Modelling and Analysis, North-Holland.

Chen, P.P (1976) *The Entity-Relationship Model - Towards a Unified View of Data*, ACM Transactions on Database Systems, **1**(1).

Codasyl (1971) Codasyl Data Base Task Group Report, ACM, New York.

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

Codd, E.F (1971) *A Data Base Sublanguage Founded on the Relational Calculus*, Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control, November 1971.

Codd, E.F (1972) *Relational Completeness of Data Base Sublanguages*, Data Base Systems, Courant Computer Science Symposia Series, **6**, Prentice-Hall.

Cuff, R. N (1982) *Database Queries Using Menus and Natural Language Fragments*, PhD Thesis, Dept of Elec Eng Science, University of Essex.

Date, C. J (1987) A Guide to INGRES, Addison-Wesley.

Date, C.J (1977) An Introduction to Database Systems, Volume I, Second Edition, Addison-Wesley.

Date, C.J (1986) An Introduction to Database Systems, Volume I, Fourth Edition, Addison-Wesley.

Deen, S.M and Hammersley, P (1981), Databases (BNCOD-1), Pentech Press, Plymouth.

Gray, P.M.D (1984) Logic, Algebra and Databases, Ellis Horwood.

Held, G.D, Stonebraker, M.R, and Wong, E (1975), *Ingres - A Relational Data Base System*, Proc. AFIPS 1975 NCC, **44**, pp.407-416.

Housel, B. C, Waddle, V and Yao, S.B (1979) *The Functional Dependency Model For Logical Database Design*, Proc. of the 5th Int. Conf. On Very Large Databases, Rio De Janeiro, Brazil.

Informix (1986) Informix-SQL Relational Database Management System, Relational Database Systems Inc. Part no. 200-41-1015-8REVB.

Johnson, S.C (1978) *Yacc: Yet Another Compiler Compiler*, in Kernighan, B.W. and McIlroy, M.D., Unix Programmers Manual, Bell Laboratories.

Katz, R.H and Goodman, N (1983), *View Processing in MULTIBASE, a Heterogeneous Database System*, in Chen (1983), pp. 257-277.

Kernighan, B. W and Pike, R (1984) The Unix Programming Environment, USA, Prentice-Hall.

Klug, A and Tsichritzis, D (1977) *Multiple View Support within the Ansi/Sparc Framework*, Very Large Databases International Conference 1977, pp. 477-488.

Kulkarni, K.G (1983) *Evaluation of Functional Data Models for Database Design and Use*, Ph.D. Thesis, University of Edinburgh.

Lacroix, M and Pirotte, A (1978) *Example Queries in Relational Languages*, Technical Note Number 107, M.B.L.E. Research Laboratories.

Lesk, M.E and Schmidt, E (1978) *Lex - A Lexical Analyser Generator*, in Kernighan, B.W and McIlroy, M.D, Unix Programmers Manual, Bell Laboratories.

Manola, F and Pirotte, A (1983) *An Approach to Multi-Model Database Systems*, in Deen and Hammersley (1981).

McDonald, N and Stonebraker, M (1975) *Cupid - The Friendly Query Language*, Proc. ACM-Pacific-75, san Francisco, Calif., April 1975, pp. 127-131.

Michaels, A.S, Mittman, B and Carlson, C.R (1976) A Comparison of the Relational and Codasyl Approaches to Data-Base Management, *Computing Surveys*, **8**(1).

Mohan, C (1978) *An Overview of Recent Database Research*, Data Base Newsletter of the SIGBDP of the ACM, 10(2).

Oracle (1986) Oracle SQL Users Guide, Oracle Corporation, Part No 3201-v1.0, Belmont, California.

Paredaens, J (1987) Databases, Academic Press.

Pratt, P.J and Adamski, J.J (1987) Database Systems: Management and Design, Boyd & Fraser, Boston.

Reisner, P (1981) *Human Factors Studies of Database Query Languages: A Survey and Assessment*, Computing Surveys, **13**(1).

Reiter, R (1984) *Towards a Logical Reconstruction of Relational Database Theory*, in Brodie *et al.* 1984.

Shipman, D.W (1981) *The Functional Data Model and the Data Language DAPLEX*, ACM Transactions on Database Systems, **6**(1), pp. 140-173.

Shneiderman, B (1980) Software Psychology, Winthrop.

Smith, J.M, Bernstein, P.A, Dayal, U, Goodman, N, Landers, T, Lin, K.W.T and Wong, E (1981a) *Multibase - integrating heterogeneous distributed database systems*, Proc. National Computer Conference, pp. 487-499.

Smith, J.M, Fox, S and Landers, T.A (1981b) Reference Manual for Adaplex, Computer Corporation of America, Cambridge, Mass.

Stocker, P.M, Gray, P.M.D and Atkinson, M.P (eds) (1984), Databases: Role and Structure, Cambridge University Press.

Stonebraker, M (1980) *Retrospection on a Database System*, ACM Transactions on Database Systems, **5**(2), pp. 225-240.

Stonebraker, M (1984) *Adding SemanticKnowledge To a Relational Database* System, in Brodie *et al* . (1984).

Stonebraker, M and Rowe, L.A (1977) *Observations on Data Manipulation Languages and Their Embedding in General Purpose Programming Languages*, Very Large Databases Int. Conf. 1977, pp. 128-143.

Stonebraker, M, Johnson, R and Rosenberg, S (1982) *A Rules System for A Relational Database Management System*, in Schauermann (1982).

Stonebraker, M, Wong, E, Kreps, P and Held, G (1976) *The Design and Implementation of Ingres*, ACM Transactions on Database Systems, **1**(3), pp. 189-222.

Taylor, R.W and Frank, R.L (1976) *Codasyl Data-base Management Systems*, Computing Surveys, **8**(1).

Tsichritzis, D.C and Klug, A (1978) *The ANSI/X3/SPARC DBMS Framework Report of The Study Group on Database Management Systems*, Information Systems, **3**, pp. 173-191.

Tsichritzis, D.C and Lochovsky, F.H (1978) *Hierarchical Data-Base Management: A Survey*, Computing Surveys, **8**(1).
Tsichritzis, D. C and Lochovsky, F.H (1983) Data Models, Prentice-Hall.

Tsur, S and Zaniolo, C (1984) *An Implementation of GEM - supporting a Semantic Data Model on a Relational Back-End*, SIGMOD RECORD, **14**(2), pp286-295.

Udagawa, Y and Ohsuga, S (1982) *Novel Technique to Interact with Relational Databases by Using a Graphics Display*, Journal of Information Processing, **5**(4).

Ullman, J. D (1982) Principles of Database Systems, Pitman.

Wiederhold, G and El-Masri, R (1979) *Structural Model for Database Design*, Proceedings of Int. Conf. on the E-R Approach to Systems Analysis and Design, North-Holland.

Zaniolo, C (1983) *The Database Language GEM*, SIGMOD RECORD, 13(4), pp.207-218.

Zloof, M.M (1977) *Query-by-Example: a database language*, IBM systems Journal, **16**(4).