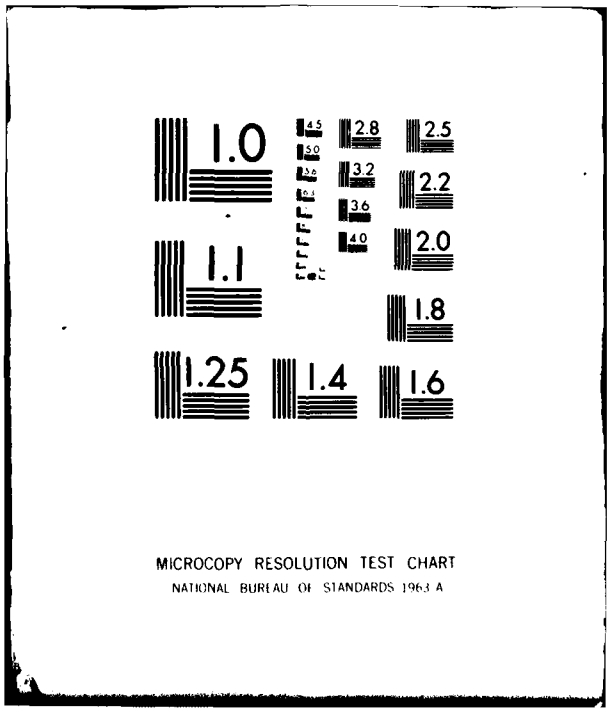
AD-A081 786 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
FQL -- THE DESIGN AND IMPLEMENTATION OF A FUNCTIONAL DATABASE Q--ETC(U)
MAY 79 R E FRANKEL N00014-75-C-0462
UNCLASSIFIED 79-05-03 NL

END
DATE FILMED
4-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

FQL--THE DESIGN AND IMPLEMENTATION OF A FUNCTIONAL DATABASE QUERY LANGUAGE

ROBERT E. FRANKEL

79-05-03

FQL -- The Design and Implementation of a Functional Database Query Language

Robert E. Frankel

79-05-03

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

This working paper reproduces in its entirety a dissertation submitted to the graduate faculty of the University of Pennsylvania, Philadelphia, PA.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) 79-05-03	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER (9)	
4. TITLE (and Subtitle) (6) FQL -- THE DESIGN AND IMPLEMENTATION OF A FUNCTIONAL DATABASE QUERY LANGUAGE.	5. TYPE OF REPORT & PERIOD COVERED Technical Report.
	6. PERFORMING ORG. REPORT NUMBER 79-05-03
7. AUTHOR(s) (10) Robert E. Frankel	8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Decision Sciences The Wharton School Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy 800 N. Quincy St., Arlington, VA 22217	12. REPORT DATE (11) May 1979 (12) 64 13. NUMBER OF PAGES 60
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

FQL (Functional Query Language), database query language, CODASYL, database management system

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

FQL is a database query language in the area of functional programming systems and has been designed and developed in an attempt to provide a powerful formalism for the expression of complex database queries. There is no notion of data currency; full computational power is provided; complex queries may be developed incrementally; and the language itself is independent of any database management system. This document describes the syntactic and semantic rudiments of FQL through a set of examples.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

L1 75

ABSTRACT

FQL is a database query language based upon recent work by John Backus in the area of functional programming systems and has been designed and developed in an attempt to provide a powerful formalism for the expression of complex database queries. As such, FQL differs from other database query languages in several important respects: there is no notion of data currency; full computational power is provided; complex queries may be developed incrementally; and the language itself is independent of any database management system. Though currently implemented with an interface to a CODASYL system, the data-model underlying the language proves sufficiently general and would allow use with other types of database systems. This document describes the syntactic and semantic rudiments of FQL through a set of examples, addresses certain issues of implementation, and considers various topics for future research and development.

CONTENTS

1.0 MOTIVATION

A basic function of any database management system is to facilitate interrogation of an integrated collection of information by a variety of users [Date 77, Mart 75]. This mode of interaction between user and database -- the query -- is generally characterized as a selective retrieval of information (possibly subject to further computation) which, nevertheless, leaves the database intact. To this end, database management systems often provide some sort of language whereby the user may formally specify his query. While often adequate for simple applications, a majority of these languages currently in use prove to be either cumbersome or else totally inadequate for the expression of complex database queries. (In support of this claim one need only consider the verbosity of the CODASYL DML on the one hand and the lack of computational power within most relational systems on the other hand.) This situation has, to a large extent, given impetus for the development of FQL, a functional database query language distinguished by virtue of its power, modularity, and precision.

The kinds of facilities presently available for retrieving information from a database management system are extraordinarily varied. At one extreme we find a class of languages best represented by the CODASYL Data Manipulation Language (DML); what essentially amounts to a collection of low-level subroutines which may be embedded in some standard

programming language and which encapsulate the more primitive search and retrieval capabilities of the database management system [Coda 71, Tayl 76]. Although one could argue for the flexibility this type of interface might afford, a major shortcoming lies with its inherently "record-at-a-time" mode of processing, as this tends to obfuscate even the simplest of queries.

This fact becomes particularly evident when one examines constructs such as FIND, the basic mechanism within the CODASYL DML for incremental (i.e., record-at-a-time) navigation through the database. Several points are worth noting: First, since mechanisms such as FIND do operate only on a record-at-a-time basis, their effective use depends upon control structures provided by a host programming language; one must explicitly account for all details of iteration and selection when, say, processing a subset of a collection of records contingent upon some condition. This lack of control abstraction within the CODASYL DML will, as a rule, promote verbosity through "procedural over-specification" -- concern with "how" to realize a particular query in lieu of "what" the query actually entails. And secondly mechanisms such as FIND, inevitably bound up with the notion of data currency, are invoked not for any value they may return but rather for the numerous effects they induce upon a global collection of pointers (currents) into the database. Not only does this type of behavior on the part of FIND shoulder the user with

additional responsibility -- if not consciously managed, the side-effects of FIND can be potentially deleterious -- but it renders queries written using the CODASYL DML virtually impossible to analyze from a formal point of view.

The alternative to the record-at-a-time mode of processing exemplified by the CODASYL DML resides in a host of query languages based largely upon the relational algebra and, to a lesser extent, upon the relational calculus [Codd 70, Codd 71, Cham 74, Cham 76, Ston 76, Zloo 75]. Unlike the CODASYL DML in several respects, the relational languages provide, as their most salient feature, an elegant set of high-level operators for manipulating entire collections of records (relations of tuples to be more precise); these operators may, moreover, be combined with one another to form powerful relational expressions having attractive mathematical properties. Since these operators implicitly subsume the details of iterative and selective control logic, not to mention the whole issue of data currency, relational languages are commonly implemented as "stand-alone" systems -- independent of any host programming language -- and, as such, have become the paradigm for interactive database query.

While the relational languages seem to rectify a number of shortcomings found in the CODASYL DML, they, too, have their drawbacks. Specifically, the relational languages are lacking in computational power; one cannot, for instance,

perform simple arithmetic in many of these systems let alone define a recursive function. (The CODASYL DML enjoys a distinct advantage in this regard by virtue of its embedding in a general-purpose programming language, though recursion can be awkward if not supported directly by the host facility.) While certain relational systems do offer some arithmetic capabilities (e.g., SEQUEL) these languages are by no means complete in a formal sense and, in fact, remain incapable of processing any sort of recursive data-structure [Aho 79].

* * *

FQL is an functional language: it is founded upon a model of computation in which the combination of functions using a pre-defined set of mechanisms, or functionals, into expressions is the only control structure. More specifically, the language embodies many of the ideas expounded by Backus concerning functional programming systems [Back 78]. Unlike conventional programming facilities, the means for manipulating an explicit data reference (i.e., the assignment of variables) has been purposely omitted from FQL. Rather, the language furnishes its user with the tools for combining a given set of primitive operators so as to form more powerful functions. As such, FQL differs from other database query languages in several important respects:

1. There is no notion of data currency: queries that operate upon entire collections of objects may be readily formulated without recourse to the "record-at-a-time" mode of processing associated with the CODASYL DML.

2. Full computational power is provided: unlike the relational languages, FQL has the capabilities of a general-purpose programming facility and, especially, supports the definition of recursive functions.

3. Complex queries may be developed incrementally: a query within FQL is but another function which, using the mechanisms furnished by the language, can be combined with other queries; in particular, FQL maintains a facility for definition of new functions thereby allowing hierarchic encapsulation of detail (something conspicuously absent from other query languages).

4. The language itself is independent of any database management system: though currently implemented with an interface to a CODASYL system, the language employs a sufficiently general data-model such that use with other types of database management systems would be possible; indeed, FQL could well serve as a common medium of communication within a network of heterogeneous database facilities.

(It should be emphasized that one should not view FQL as the "ideal" end-user query language; such a language, in the opinion of this author, does not exist. Rather, it is presented as a precise and powerful formalism that can serve both as a tool for those wishing to construct complex database queries and as an intermediate language into which one may readily translate his "favorite" query protocol.)

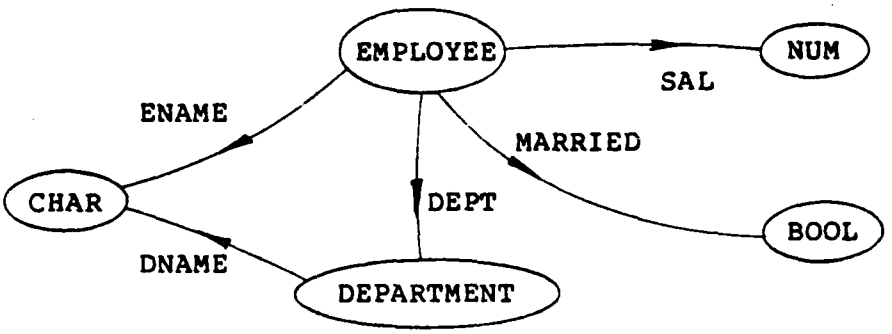
The remainder of this document is devoted to a detailed explication of FQL. Section 2.0 describes the syntactic and semantic rudiments of the language: beginning with a brief exposition of FQL's underlying data-model, the various

mechanisms afforded by the language for combining functions together with simple examples of their application are then presented; more sophisticated queries are discussed subsequently. Section 3.0 addresses certain issues of implementation, summarizes the more important internal data- and control structures, and traces the system's execution of a simple query. (In its current form, FQL is implemented in PASCAL as an interface to SEED [Gerr 78], a CODASYL-based database management system written in FORTRAN, and is running on a DEC-10.) Finally, section 4.0 considers various areas for future research and development.

2.0 THE LANGUAGE

2.1 A Functional Data-Model

Since FQL provides its user with only the ability to combine functions, the language assumes, for its underlying data-model, a functional view of databases. That is to say, we regard a database as comprising a collection of functions over various data-types [Ship 79]. To illustrate, consider the functional schema of a (very simple) database built around entities of type EMPLOYEE and of type DEPARTMENT:



The nodes of this graph denote the particular data-types present within the database while its directed edges represent mappings between these types; thus given an EMPLOYEE the function DEPT would return that DEPARTMENT, say, in which he works. In addition to providing a function from EMPLOYEE to DEPARTMENT, this database furnishes operators that map these entities into the familiar, basic types: the functions ENAME and DNAME each return a CHAR(acter string); the function SAL(ary) yields a NUM(eric) value; and the BOOL(ean) function MARRIED serves as a predicate. Using a more conventional notation we

summarize the functional schema of this database as follows:

```
DEPT     : EMPLOYEE   -> DEPARTMENT
ENAME    : EMPLOYEE   -> CHAR
SAL      : EMPLOYEE   -> NUM
MARRIED  : EMPLOYEE   -> BOOL
DNAME    : DEPARTMENT -> CHAR
```

Of the five data-types seen here, only the types CHAR, NUM, and BOOL are considered standard in that they exist independently of any particular database schema. The types EMPLOYEE and DEPARTMENT, together with the functions defined over them, are, on the other hand, specific to this database. We should, at this time, also mention that these latter types are not to be regarded as structured (in the sense of a record or tuple) but rather as scalar; thus an EMPLOYEE is no less atomic than the NUM(ber) denoting his salary. Unlike the standard scalars, though, one cannot speak of printing values of type EMPLOYEE or of type DEPARTMENT per se. Information about these entities may, however, be obtained through application of those functions which do map these objects into some "printable" type (e.g., CHAR, NUM, or BOOL).

(Viewing entities such as EMPLOYEE or DEPARTMENT as scalars should not, of course, imply anything about their physical representation within a particular database management system; the functional model is, after all, only intended as an abstraction. Indeed, when imposing a functional view upon a CODASYL system, types such as

EMPLOYEE and DEPARTMENT are, in fact, the records within a schema; the items these records contain become, in turn, the functions which map these database-dependent types into the standard scalars.)

We have, until now, considered only those functions provided by the database which map scalars into other scalars. To deal with operators that produce, say, the set of employees who work in a given department, we must augment our functional view of databases to include the inverse of functions as well. Knowing, for instance, that the function DEPT takes an EMPLOYEE into his DEPARTMENT, its inverse, written !DEPT, would map a DEPARTMENT into the sequence of EMPLOYEEs belonging to it; the function !DNAME would, as another example, map a CHAR(acter string) into an ordered collection of DEPARTMENTs, each of which bear the same name. (Strictly speaking, !DEPT and !DNAME are not the true mathematical inverses of DEPT and DNAME respectively, since the inverse of a function should contain no notion of sequentiality.) In referring to these sequences of homogeneously-typed objects we shall use the more abstract term stream; a stream, following Landin [Land 65] and Burge [Burg 75], is a "virtual" sequence of objects whose physical representation should be of no concern to its processor -- it may be a list in primary store, a file in secondary store, a generating function, or even some combination of these. Again, we summarize using conventional notation, where type specifications prefixed

with a "*" denote a stream of such objects:

```
!DEPT   : DEPARTMENT -> *EMPLOYEE
!DNAME  : CHAR -> *DEPARTMENT
```

(It should be stressed that the "!" in no way constitutes a general "inversion" operator but rather serves as a notational convention. Whether or not the inverse of a particular function is available depends entirely upon the database: there is no guarantee that, because the function SAL exists, its inverse, !SAL, will also be present. Database systems do, though, usually employ sophisticated mechanisms for implementing the inverses of functions when they are required. The function !DEPT, for example, could be realized by a CODASYL set whose owner is a DEPARTMENT record and whose members are EMPLOYEE records; the function DEPT itself is suggested by the linkage between each member of the set and its unique owner. Likewise, the designation of particular items within records as calc keys (descriptors) gives rise to functions such as !DNAME which effectively provide direct access, usually by means of a hash table, to certain records within a file.)

Another means for gaining access to collections of objects is through a number of constant functions which yield sequences comprising all instances of each database-dependent scalar type currently in the system. (A constant function is one whose value is independent of its argument.) In our sample database these include the constant

functions !EMPLOYEE and !DEPARTMENT which return streams whose elements would, respectively, be of type EMPLOYEE and of type DEPARTMENT. (Of course, the functions !EMPLOYEE and !DEPARTMENT are not truly constants, for not only are they database-dependent but their values can, and often do, change over the course of time; as far as their actual implementation is concerned, functions such as these are simply the "master" files within any database management system.) In summary, then:

```
!EMPLOYEE   : -> *EMPLOYEE
!DEPARTMENT : -> *DEPARTMENT
```

(The absence of a type specification to the left of the "->" serves to indicate a constant mapping.) As we shall soon see, queries are but a special kind of constant -- albeit database-dependent -- function.

(FQL supports, in addition to streams, one other mode for structuring information -- the tuple. Since, though, our functional data-model precludes the existence of tuples within the database we have not dealt with them at this time; tuples do, though, play an important role within the language and will be treated subsequently.)

2.2 Mechanisms For Combining Functions

Given that the database comprises a collection of functions over various data-types, we now need

mechanisms -- functional forms as Backus terms them -- for combining these primitive operators in order to create new and more powerful functions; and ultimately to formulate queries. FQL provides four such functional forms -- composition, extension, restriction, and construction -- that are individually exemplified below using our database of EMPLOYEEs and DEPARTMENTs presented in section 2.1. So as to facilitate the use of these mechanisms, a simple means for function definition has been included within the language. In general, the format for definition of a new function within FQL is as follows:

<name>: <input type> -> <output type> =
<functional expression> ;

In words, each definition associates an arbitrary name with a functional expression such that any occurance of the former within some other expression may be replaced by the latter. Each definition also specifies the new function's input- and output type which must concord with the domain and range implied by its body.

* * *

Composition

Among the functional forms furnished by FQL, the most fundamental is composition. Intuitively, composition is a coupling of two functions such that the output of one serves

as input for the other. One can, for instance, compose the operator DEPT (a mapping from EMPLOYEE to DEPARTMENT) with the operator DNAME (a mapping from DEPARTMENT to CHAR) so as to form a functional expression which, given an employee, returns the name of that department in which he works. (A functional expression in general is the composition of one or more functions.) In this way, we may define a new function called DEPTNAME:

```
DEPTNAME: EMPLOYEE -> CHAR =
    DEPT.DNAME ;
```

DEPTNAME is here declared as a mapping from EMPLOYEE to CHAR(acter string) and is defined as the composition of DEPT with DNAME, denoted by the operator ".". As another example, let us define the function COWORKERS that returns, for a given employee, those employees with whom he works:

```
COWORKERS : EMPLOYEE -> *EMPLOYEE =
    DEPT.!DEPT ;
```

We should, at this time, point out that all functions are composed, and hence evaluated, from left to right (reverse Polish); within the body of COWORKERS the DEPARTMENT gotten by applying DEPT to some EMPLOYEE becomes the operand for !DEPT which, in turn, produces as its value a stream of employees (i.e., *EMPLOYEE). (Using reverse Polish for functional composition in fact seems quite natural when confronted with a database, as the left to

right order of evaluation actually determines a corresponding path through the database schema.) To summarize more formally: if f and g are functions such that $f: \alpha \rightarrow \beta$ and $g: \beta \rightarrow \gamma$ then their composition, denoted f.g, is a mapping of the form $\alpha \rightarrow \gamma$, where Greek letters signify arbitrary data-types. In general, then, a functional expression of the form $f_1.f_2. \ldots .f_n$ accepts, as input, a value from the domain of function f_1 and produces, as output, a value from the range of function f_n.

Extension

Extension is one of two functional forms that specifically map streams into other streams. Roughly speaking, extension is a for each operator in that it allows for the uniform processing of some collection of objects. To consider an example, the function SAL (a mapping from EMPLOYEE to NUM) may be "extended" into a function written *SAL such that, given a stream of employees (*EMPLOYEE), it returns a stream of their salaries (*NUM) by applying the function SAL to each EMPLOYEE in turn. (The use of the "*" here in describing both functions and types is intended to underscore the similarity between them.) Needless to say, a function such as *SAL may be composed with yet other functions in order to form expressions. Thus, one can define a function DEPT-SALS that yields the salaries of each employee within a particular department:

```
DEPT-SALS: DEPARTMENT -> *NUM =
    !DEPT.*SAL ;
```

Here, the result of applying !DEPT gives a stream of EMPLOYEEs over which the function SAL is then extended to produce a stream of NUMs. Indeed, we may even "extend" this particular function so that, given a stream of departments (*DEPARTMENT) it would return a stream of streams of their workers' salaries (**NUM). In summary then: if f is a function such that f: α -> β then its extension, denoted *f, is a mapping of the form *α-> *β. Note that the arbitrary function, f, being extended is in no way limited to primitive operators or the names of other functions defined previously but may itself be an explicit instance of further functional combination.

Restriction

Restriction, like extension, maps streams into other streams. Unlike extension, though, which preserves the length of its operand (but possibily altering the type of its components), restriction will always return the same type of stream as given though generally with fewer elements. Specifically, restriction filters a stream by a predicate defined over a typical element: the function !MARRIED (where "!" signals restriction) would therefore map a stream of EMPLOYEEs into a sub-stream of EMPLOYEEs satisfying the condition that they be MARRIED. We may,

then, restrict the result of our earlier function COWORKERS in the following manner:

```
MARRIED-COWORKERS: EMPLOYEE -> *EMPLOYEE =
    COWORKERS.|MARRIED ;
```

And in general: if p is a predicate such that p: α-> BOOL then the restriction of a stream *α, denoted |p, is a transformation * α-> *α. Again, the predicate need not be simple though, for the moment, we lack the tools for forming more complex boolean expressions.

Construction

As mentioned at the end of section 2.1, FQL supports the structuring of data into tuples. A tuple, complementing the notion of a stream, is a heterogeneous aggregation of some fixed number of objects. Construction allows for the collateral application of a number of functional expressions to a common argument, thereby yielding a tuple of their respective results. In describing these tuples we would, as an example, use the notation [CHAR,NUM] to denote the data-type of tuples of type CHAR and NUM. This type of structure would arise if, for instance, one wished to produce the name and salary of a given employee. Using construction, this is realized by the function [ENAME,SAL]. (Again, observe the conformity in notation between function and type.) As a more substantial example, let us define a function that returns both the name of a particular

employee's department together with the salaries of his co-workers:

```
DNAME-AND-MARRIEDWORKERS-SALS: EMPLOYEE -> [CHAR,*NUM] =
    DEPT.[DNAME,!DEPT.|MARRIED.*SAL] ;
```

Analyzing this definition, the function DEPT produces a DEPARTMENT which, in turn, serves as an operand for both the operator DNAME and the mapping that results from composing the restriction of !DEPT with the extension of SAL (i.e., DEPARTMENT->*NUM), thereby constructing a tuple comprising both a department name and a stream of salaries. By way of summary: if f_1, f_2, ... f_n are functions such that $f_1: \alpha \rightarrow \beta_1$, $f_2: \alpha \rightarrow \beta_2$, ... $f_n: \alpha \rightarrow \beta_n$ then their construction, denoted $[f_1, f_2, \ldots f_n]$, is mapping of the form $\alpha \rightarrow [\beta_1, \beta_2, \ldots \beta_n]$. Though tuples are indeed useful for extrapolating a multiplicity of data about entities such as EMPLOYEEs or DEPARTMENTs, they will become increasingly important when, for instance, the standard arithmetic operators are introduced; since, by design, all FQL functions (primitive or composite) are monadic, the function "plus" is taken as a mapping from a pair of numbers into a single number; e.g., +: [NUM,NUM]->NUM.

* * *

A query in FQL is a constant function whose result is a value of some "printable" type, where the type of a printable object is recursively defined as either that of a standard scalar, a tuple of printables, or a stream of such. As for a query being a constant mapping, this implies it be the composition of one or more functions such that the first of these is a constant operator (either primitive or the construction of yet other constants). By way of example, consider a query which simply gives "the name of each department currently in the database":

```
Q1: -> *CHAR =
        !DEPARTMENT.*DNAME ;
```

(Again, the absence of an input-type specification denotes a constant mapping.) As a more complex example, consider a query which returns "for every employee the name of his department together with the salaries of his married coworkers". Using a function defined previously, the solution follows trivially:

```
Q2: -> *[CHAR,*NUM] =
        !EMPLOYEE.*DNAME-AND-MARRIEDWORKERS-SALS ;
```

The point to be made here is that complex tasks may be solved incrementally within FQL through a hierarchy of function definitions, each encapsulating some lower level of detail; certainly, at the highest level, query Q2 is no more difficult to comprehend than the simpler query Q1.

Were, though, a mechanism for the definition of new functions not included within the language, the formulation of query Q2 would indeed appear more complicated:

```
Q2': -> *[CHAR,*NUM] =
        !EMPLOYEE.*(DEPT.[DNAME,!DEPT.|MARRIED.*SAL]) ;
```

(Note that parentheses must be used here to enforce the order of evaluation.)

To recapitulate, we have viewed a database as a collection of functions over various data-types and have presented four functional forms -- composition, extension, restriction, and construction -- for combining these functions into new functions, and finally into queries; we have also introduced two modes for structuring data -- streams of the form $*\alpha$ and tuples of the form $[\beta_1, \beta_2, \ldots, \beta_n]$. By way of summary, the FQL syntax of a function definition, a data-type, a functional expression, and a function itself is given at this time:

```
<def> ::= <name>: {<type>}' -> <type> = <fexpr> ;

<type> ::= NUM
       ::= CHAR
       ::= BOOL
       ::= *<type>
       ::= [<type>{,<type>}+ ]

<fexpr> ::= <function>{.<function>}
```

```
<function> ::= <primitive>
           ::= <name>
           ::= *<function>
           ::= |<function>
           ::= [<fexpr>{,<fexpr>}+ ]
           ::= (<fexpr>)
```

(Optional components are denoted by "{"..."}$^{!}$ " while "{"..."}$^{+}$ " signifies that a set of elements may occur an arbitrary number of times.)

2.3 Standard Functions

The class of queries one can formulate using only those functions provided by the database is rather limited. To extend its computational power FQL furnishes an array of standard -- database independent -- functions including the familiar arithmetic, relational, and logical operators together with a host of primitives for manipulating structured data-types. These various functions are introduced below, grouped into categories as appropriate. (It should be mentioned that the following listing is in no way complete or comprehensive; rather, it represents a selection of useful primitives through which more complex functions may, if needed, be realized. The reader may, if he so chooses, omit this material for the time being and refer back to it when necessary.)

* * *

Arithmetic Operators

The functions +, -, x, /, and MOD are each mappings from the pair [NUM,NUM] into NUM. The functions /+ and /x perform addition- and times-reduction on streams of NUMs; i.e., they map *NUM into NUM. Given an empty stream these functions return their respective identities, 0 and 1.

Relational and Boolean Operators

The operators EQ, NE, GT, LT, GE, and LE map either tuples of the form [NUM,NUM] or of the form [CHAR,CHAR] into a single BOOL(ean) value. The functions AND and OR each return a BOOL given a [BOOL,BOOL] pair; the complement NOT takes a single BOOL into another BOOL. The two reduction operators, /OR and /AND, represent mappings from *BOOL to BOOL and, given empty streams, return the values "true" and "false" respectively.

Constants

The notation #<number> represents a constant mapping of the type ->NUM whose value is the <number>; the notation '<character-string>' similarly denotes the mapping ->CHAR. The function NIL is a constant signifying the empty stream of any type; i.e., ->*α.

Basic Stream-Manipulating Primitives

Given a non-empty stream, the operation HD returns its first element (i.e., *α->α) while the operation TL returns a sub-stream containing all but its first element (i.e., *α->*α). The function CONS takes an element of some type and a (possibly empty) stream whose elements are of that same type and returns a new stream in which the individual element is its "head" while the original stream becomes its "tail"; i.e., CONS : [α,*α]->*α.

Other Stream-Manipulating Primitives

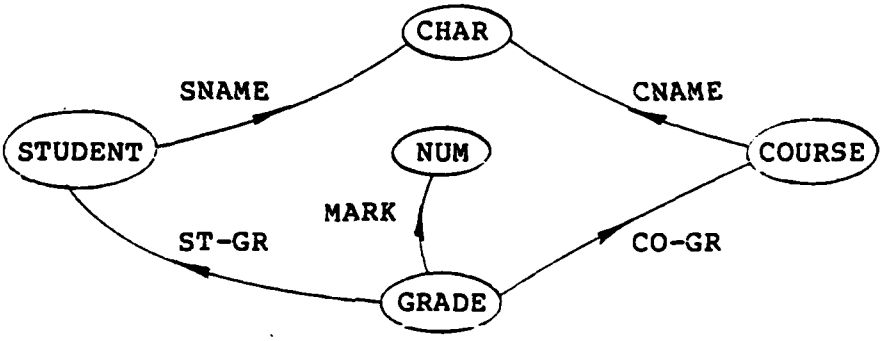
The function LEN computes the length of a given stream and is thus a mapping from *α into NUM. CONC concatenates a pair of streams [*α,*α] (whose elements are of the same type) into a single stream *α; /CONC produces a single stream *α by "flattening" an arbitrary stream of streams **α. The operator DISTRIB takes a tuple of the form [*α,β] and returns a stream of tuples *[α,β] with the value of type β effectively paired with each element within the stream of α's.

Miscellaneous Functions.

The function i (i=1,2,...n) selects a component from an arbitrary n-tuple; i.e., i: $[\beta_1, \beta_2, \ldots \beta_n] \rightarrow \beta_i$. Finally, ID represents the identity mapping $\alpha \rightarrow \alpha$.

* * *

In order to demonstrate the use of a number of these standard functions, we shall consider several queries over an academic database containing STUDENTs, COURSEs, and collections of their respective GRADEs. The schema of this database is given by the following:




```
SNAME    : STUDENT -> CHAR
CNAME    : COURSE  -> CHAR
MARK     : GRADE   -> NUM
ST-GR    : GRADE   -> STUDENT
CO-GR    : GRADE   -> COURSE
!CNAME   : CHAR    -> *COURSE
!ST-GR   : STUDENT -> *GRADE
!CO-GR   : COURSE  -> *GRADE
!STUDENT:          -> *STUDENT
!COURSE  :          -> *COURSE
!GRADE   :          -> *GRADE
```

Basically, the functions ST-GR and CO-GR provide the means for mutual association of STUDENTs and COURSEs through their common GRADEs: the expression !ST-GR.*CO-GR returns a given student's courses while the expression !CO-GR.*ST-GR

produces the students enrolled in a particular course.

As an initial example, we wish to know "the average grade of each student". First, though, let us define the function AVRG which computes the mean of a stream of numbers:

```
AVRG: *NUM -> NUM =
      [/+,LEN]./ ;
```

The function /+ sums the elements of the given stream while LEN returns its length; this pair of NUM(bers) then serves as the dividend and divisor for the division operator. The query itself is expressed by:

```
Q3: -> *NUM =
      !STUDENT.*(!ST-GR.*MARK.AVRG) ;
```

Here, the operand for AVRG is produced by extending the operator MARK over each individual student's grades; this mapping from STUDENT to NUM(ber) in turn is applied to each student currently in the database.

The next query illustrates the formulation of boolean expressions using the relational and logical operators within the language. Specifically, it asks for "the names of those courses with an enrollment between 10 and 50 in which some student received a mark below 70". In FQL:

```
Q4: -> *CHAR =
        !COURSE.|([P1,P2].AND).*CNAME ;
```

The query is formed through restricting the stream of all COURSEs by the conjunction of predicates P1 and P2 as specified below (again, observe the use of parentheses to enforce the scope of the restriction operator); the function CNAME is then extended over the resulting sub-stream. The predicates P1 and P2 on individual COURSEs are defined as follows:

```
P1: COURSE -> BOOL =
        !CO-GR.LEN.[[ID,#10].GE,[ID,#50].LE].AND ;

P2: COURSE -> BOOL =
        !CO-GR.*([MARK,#70].LT)./OR ;
```

The first predicate initially counts the enrollments in the given course -- the stream produced by !CO-GR; using the identity operator this number is then propagated through the ensuing conjunction in which it is compared with the values 10 and 50 (more precisely, with the values of the constant functions #10 and #50). The second predicate tests for the presence of some MARK below 70: given the GRADES within the COURSE, the expression *([MARK,#70].LT) yields a stream of BOOL(eans) with the operator /OR returning a value of "true" if some member of this stream is "true".

As a final example we take up the following task: "the number of students enrolled in both ENGLISH and HISTORY". (The trick here, of course, is not to count a student twice.) To facilitate a solution we first introduce a function that, given the name of a course, returns the name of each student enrolled within it:

```
CLIST: CHAR -> *CHAR =
     !CNAME.HD.!CO-GR.*(ST-GR.SNAME) ;
```

Observe that while !CNAME produces a stream of courses one is, in general, only interested in the first of these, thus the operator HD must be applied; the names of the students within this course are then extracted via its stream of individual GRADEs. The query itself may be expressed as follows:

```
Q5:  -> NUM =
     ['ENGLISH'.CLIST,'HISTORY'.CLIST].INTERSECT.LEN ;
```

(Note here that the query is still a constant function by virtue of the fact that each expression within the initial usage of the construction mechanism is itself a constant.) Ideally, set-theoretic operators such as INTERSECT should be furnished by FQL. Since (at present anyway) these operators are not supported we must resort to the following definition:

```
INTERSECT: [*CHAR,*CHAR] -> *CHAR =
     DISTRIB.|MEMBER.*1 ;
```

The second stream is distributed over each of the elements of the first stream, giving a structure of the form *[CHAR,*CHAR]. The predicate MEMBER (defined below) returns a value of true for a typical element of this stream of pairs if its first component (CHAR) is to be found anywhere within its second component (*CHAR); the extension of the selector "1" effectively projects the first components of those pairs satisfying the previous predicate. The function MEMBER is then defined as follows:

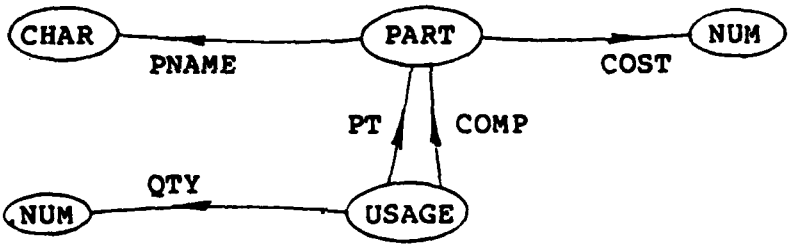
```
MEMBER: [CHAR,*CHAR] -> BOOL =
        [2,1].DISTRIB.*EQ./OR ;
```

The expression [2,1] effectively reverses its operand so that DISTRIB may pair the individual CHAR with each element within the given stream of CHARs; the expression *EQ./OR returns a value of "true" if one of these pairs satisfies the equality relation.

2.4 A Bill-of-Materials Processor

As a further demonstration of the power of FQL, we will attack the infamous bill-of-materials problem which, to this author's knowledge, eludes solution (or at least an elegant one) within most database query systems. The difficulty here lies with the fact that the schema required is inherently recursive: parts contain sub-parts which themselves are parts; these, in turn, are built out of

other parts; and so on. The specific task addressed initially is that of finding "the total cost of a given part". If we associate with each part a "cost" meaning either the purchase price (in which case it has no sub-parts of interest to us) or else the expense of assembly, then the total cost of a part is its own "cost" added to the total cost of all of its sub-parts. To complicate matters somewhat we will assume the components of parts are used in differing quantities: an engine, for example, may require four piston assemblies and two carburetors. The following depicts a possible schema for such a database:




```
COST    : PART  -> NUM
QTY     : USAGE -> NUM
PNAME   : PART  -> CHAR
PT      : USAGE -> PART
COMP    : USAGE -> PART
!PNAME  : CHAR  -> *PART
!PT     : PART  -> *USAGE
!COMP   : PART  -> *USAGE
!PART   :       -> *PART
!USAGE  :       -> *USAGE
```

The relation between a part and its sub-parts is represented by the USAGE type. For example, if an engine requires two carburetors, a USAGE entity will be defined whose PT is an engine, whose COMP is a carburetor, and whose

QTY is 2. The expression !PT.*COMP therefore maps a given PART into a stream of its immediate COMP(onents); conversely, !COMP.*PT returns a stream of PARTs in which a given PART is an immediate constituent. We may now define the (recursive) function TC which computes a part's total cost:

```
TC: PART -> NUM =
      [!PT.*([COMP.TC,QTY].x)./+,COST].+ ;
```

For a given PART, the total-cost (TC) of each of its sub-parts is multiplied by the required quantity and then summed together, after which this total is added to the COST of the original PART.

What is remarkable about this particular function is that its definition, despite recursion, includes no explicit basis for termination. (The IF-THEN-ELSE construct normally associated with termination in recursive functions is generally not required within FQL.) Yet, computation will halt since the database is finite. This can be seen by examining the simplest case: given an atomic part (one with no sub-parts), application of the function !PT would yield the empty stream; applying the parenthesized expression to each element of this stream produces, of course, another empty stream; the sum of the empty stream of NUMs is, by definition, 0 (the identity for addition) which is then added to the cost of the original part. And assuming the components of all parts are ultimately atomic the function

TC will converge. Incorporation of the function into an appropriate query is left to the reader's imagination.

The next task considered here draws upon several of FQL's stream-manipulation primitives. Simply, we wish to know "the bill of materials for a given part". One solution to this problem essentially involves a depth-first traversal of the "part-tree" within the database as shown by the following:

```
DFBOM: PART -> *USAGE =
    !PT.*([ID,COMP.DFBOM].CONS)./CONC ;
```

Here, each of the USAGEs returned by !PT for the given PART is CONSed onto the front of the bill-of-materials of that particular COMP(onent) PART to which it refers; the result, a stream of streams of USAGEs, is subsequently flattened by /CONC. Again, this function need not contain any explicit test for terminal conditions: if the stream produced by !PT is empty for a particular part, the extension of the interior expression gives another empty stream which, in fact, serves as the identity for /CONC. The function DFBOM may then be used to retrieve the name and quantity of each part within, say, an ENGINE:

```
Q6: -> *[CHAR,NUM] =
    'ENGINE'.!PNAME.HD.DFBOM.*[COMP.PNAME,QTY] ;
```

Complementing the solution of this problem through the function DFBOM one could perform a breadth-first traversal

of (in general) a forest of "part-trees" as well. Consider:

```
BFBOM: *PART -> *USAGE =
    *(!PT.[ID,*COMP.BFBOM].CONC)./CONC ;
```

(It is indeed interesting to note the inherent similarity between depth- and breadth-first traversal as expressed in FQL; verification of BFBOM's termination is left as an exercise for the reader.) The query, then, must be slightly modified:

```
Q6': ->*[CHAR,NUM] =
    'ENGINE'.!PNAME.BFBOM.*[COMP.PNAME,QTY] ;
```

3.0 IMPLEMENTATION

3.1 System Architecture

As the manipulation of streams figures prominently in a majority of database queries, a fundamental aspect of the design and implementation of an FQL processor is supporting the user's perception of continually traversing and constructing (possibly very long) lists of objects, though without actually using large amounts of primary or secondary store. Consider, as an example, a query that yields "the department name and salary of each married employee", given the database described in section 2.1:

```
!EMPLOYEE.|MARRIED.*[DEPT.DNAME,SAL]
```

Ignoring, for the moment, issues of efficiency one best comprehends the semantics of this expression in terms of a succession of discrete list-processing operations: !EMPLOYEE generates a physical sequence of all employees currently in the database; restriction by the predicate MARRIED effectively causes a complete traversal of this list, producing as its result a new sequence of employees; subsequent extension in turn entails a list traversal and creates yet another sequence (one comprising CHAR-NUM pairs); this final list is then implicitly output.

While useful conceptually in analyzing the manipulation of streams within an FQL query, the metaphor of literally mapping one collection of objects into another would

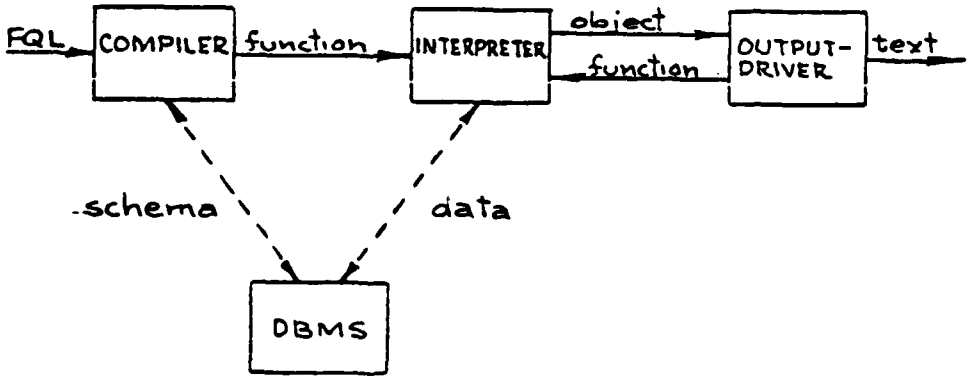
certainly prove untenable as a basis for an underlying implementation. (Many implementations of the relational languages do, in fact, faithfully adhere to this metaphor; and performance suffers accordingly due to the overhead incurred through creation and traversal of temporary files.) From the point of view of efficiency, the query posed above ought to be realized more or less along the following lines:

```
REPEAT
  find and get next employee;
  IF married THEN BEGIN
    find and get employee's department;
    print department-name and employee-salary
    END
UNTIL end-of-file
```

Though in principle one could translate functional expressions within FQL into some standard statement-oriented formalism, the compilation process becomes extremely complex particularly when confronted with explicit use of the language's various "list-processing" primitives -- CONS for instance -- in conjunction with recursive functions (as in the "bill-of-materials" problem sketched in section 2.5).

The architecture of the FQL processor described subsequently has, by in large, been influenced by the work of Friedman and Wise who have pursued various techniques for ameliorizing the operational semantics of a class of applicative list-processing languages (e.g., pure LISP) [Frie 76a, Frie 77]. (There are, though, a number of significant differences which improve the system's overall performance as a database query facility; for one, all of

the stream-manipulating operators together with functionals such as extension and restriction have been implemented directly, though in fact many of these may be defined in terms of a more basic set of primitives.) Following Friedman and Wise, the FQL processor is essentially an output-driven interpreter for expressions whose results, if structured, have computation of their individual components "suspended" until needed by the system. As a consequence, many queries apparently necessitating large amounts of intermediate store (cf. our previous example) will in fact be realized through a single file traversal with a minimal storage overhead in which input/output is effectively overlapped with computation. The following gives a more complete picture of the system as a whole:



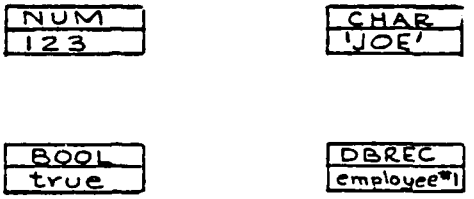
The compiler produces, as its "object code", an internal representation of an FQL functional expression while performing the usual syntactic and semantic analysis; the interpreter evaluates this expression and yields an internal

object to be printed by the output-driver. (A resident database management system furnishes, upon request, both information pertaining to the schema along with the actual data of some specific database.) Of particular note is the fact that the output-driver may itself feed the interpreter with an internal representation of a functional expression; in general, these two modules will operate in tandem as a pair of co-routines.

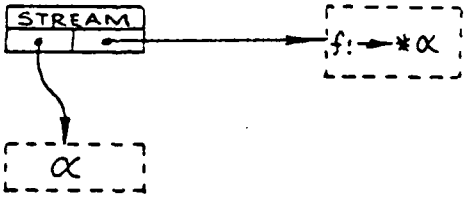
3.2 Internal Data And Control Structures

Objects

Roughly speaking, the internal data-structure used by the FQL processor to represent an object -- what the user sees as a scalar, stream, or tuple -- contains two fields respectively indicating its kind and value. A standard scalar, for example, would have NUM, CHAR, or BOOL as its "kind" and an integer, character string, or bit as its "value"; in the present implementation which interfaces with a CODASYL database management system a database-dependent scalar (e.g., an EMPLOYEE) is of "kind" DBREC and has, for its "value", a reference to some physical record in secondary store. Graphically, the internal structure of these various types of scalar objects may be viewed as such:

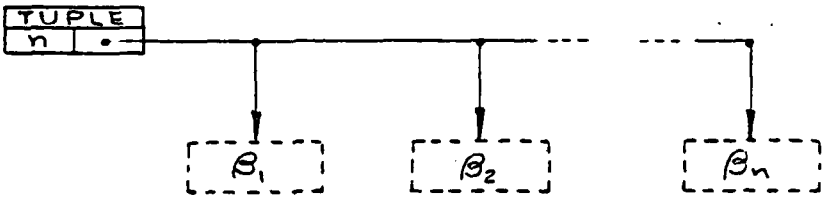


Streams are represented within the FQL processor as objects whose "value" is further decomposed into two fields -- a head which is another object representing a typical element of the stream and a tail consisting essentially of a constant functional expression which, when applied, produces another internal stream object. Given an arbitrary stream whose elements are either scalars, tuples, or themselves streams (i.e., $*\alpha$), the following would depict its internal format:

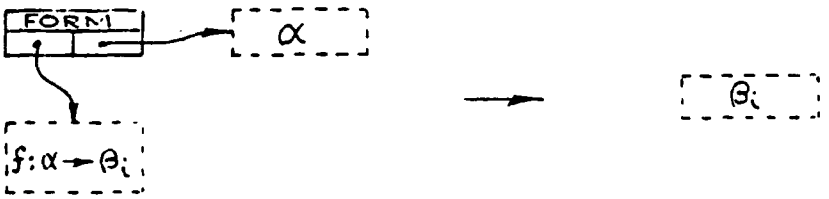


By "suspending" generation of the stream's tail in this manner very long (indeed infinite!) sequences may be efficiently managed.

Finally, a tuple is represented within the FQL processor as an object whose "value" includes both its length, an integer, and its components, an array of objects. Thus:



We should, though, point out that a typical component, β_i, $(1 \leq i \leq n)$ is initially represented as a form, a special kind of object whose "value" comprises a fun and an arg, respectively a functional expression and an object; only when this particular member of the tuple is germane to the computation at hand is this form actually replaced by the object it denotes -- a scalar, stream, or tuple. Thus the following transformation occurs when selecting the ith component of an arbitrary n-tuple for the first time:



To review the various aspects of objects and their internal representation within the FQL processor it is instructive to consider the procedure PRINT, the system's output-driver. First, though, let us summarize the structure of an FQL object using a PASCAL-like formalism:

```
TYPE object = RECORD
  CASE kind OF
    num    : (value : integer);
    char   : (value : character_string);
    bool   : (value : boolean);
    dbrec  : (value : /* secondary storage address */);
    stream : (head : object ;
              tail : LIST OF function);
    tuple  : (length : integer ;
              components : ARRAY 1..length OF object);
    form   : (fun : LIST OF function ;
              arg : object)
  END
```

(Note that "LIST OF function" -- a departure from standard PASCAL -- here represents a functional expression.) The procedure PRINT is itself responsible either for printing the value of some standard scalar or, given a structured object, for (recursively) printing the values of its various elements. Again, the details are specified using a PASCAL-like formalism:

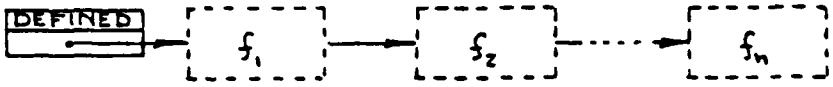
```
PROCEDURE print (ob : object);
  BEGIN
    CASE ob.kind OF
      num,char,bool : write (ob.value);
      stream : BEGIN
        write ('{');
        WHILE not empty (ob) DO BEGIN
          print (ob.head);
          ob := apply (ob.tail,Ω)
          END;
        write ('}')
        END;
      tuple : BEGIN
        write ('[');
        FOR i := 1 TO ob.length DO BEGIN
          eval (ob.components[i]);
          print (ob.components[i])
          END;
        write (']')
        END
      END
  END
```

Note that when given a stream the output-driver, after PRINTing the current "head", must repetitively invoke the interpreter via the system function APPLY (described below) in order to generate the remaining elements within sequence; the operands to APPLY in this case include the current stream's "tail" -- a constant mapping -- together with a value, Ω, of some arbitrary anonymous object. Likewise, when given a tuple PRINT calls upon the system procedure EVAL to coerce each component prior to output.

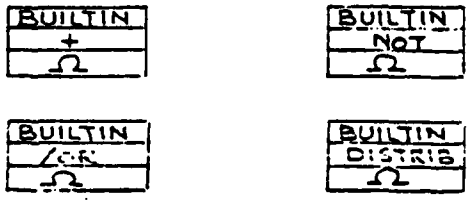
Functions

The internal representation of a function within the FQL processor (like that of an object) is tagged according to its kind. On the one hand, a function may be DEFINED and therefore has a definition, a functional expression denoted

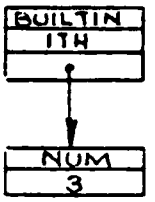
by a list of other functions. Graphically, the body of a function (g) defined as the composition of functions f_1, f_2 ,... f_n may be depicted as follows:



On the other hand, a function may be BUILTIN in which case it has both an opcode and an environment, the latter being a list of objects. With only a few exceptions, though, the "environment" field of FQL's various standard functions remains empty. For instance:



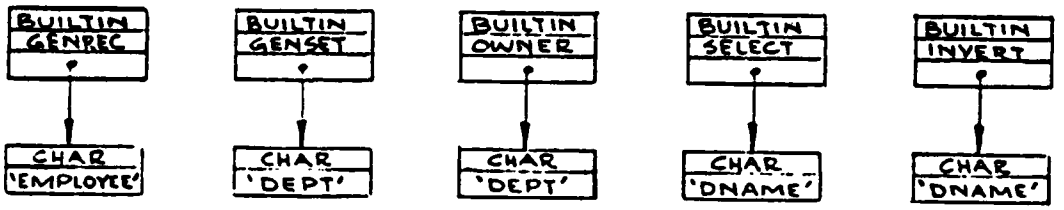
The "environment" is, however, necessary when representing a function that selects, say, the third component of a given tuple (i.e., the operator "3"). Consider:



The function ITH (the canonical selector for tuples) takes two operands, one of which is an integral index capable of being bound or partially applied at "compile time". A similar situation arises when representing the standard constants: the FQL functions "#123" and "'JOE'" are internally denoted by the operator CONSTANT whose environment would respectively contain an INT and a CHAR.

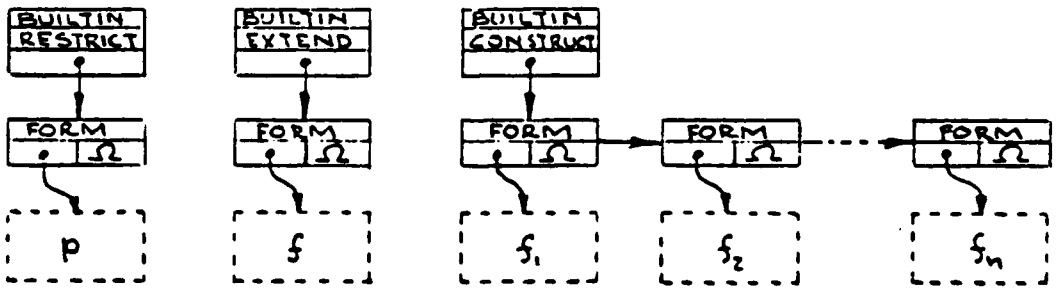
This notion of generic operations, some of whose parameters may be bound prematurely, is particularly prevalent in the representation of database-dependent functions. With the present CODASYL implementation, the system internally utilizes five such operators: GENREC which generates a stream of all current instances of a particular record-class; GENSET which generates a stream of all members within a particular set, given the owner record; OWNER which returns the owner of a particular set, given some member record; SELECT which retrieves a particular item from a given record; and INVERT which produces a stream of records containing a given key. Returning, again, to the database of section 2.1, the following are the internal representations of the functions !EMPLOYEE, !DEPT,

DEPT, DNAME, and !DNAME:



Here, all database-dependent information has been subsumed within the environments of a more general set of operators.

Of the four functional forms provided by FQL only the composition of functions into expressions is indicated implicitly (a functional expression, as noted above, is represented as a list of functions). The other functionals are denoted explicitly by the operators RESTRICT, EXTEND, and CONSTRUCT whose environments would (indirectly) contain other functions as their parameters. The following is the internal format for both restriction and extension by some predicate or function and for construction of an arbitrary n-tuple:



Note that although the objects comprising the environments of these functions are of "kind" FORM, the "arg" field in general will not be used.

To round out our discussion of functions and their internal representation within the FQL processor, we shall focus briefly upon APPLY and EVAL, the major components of the system's interpreter. First, though, let us summarize the format of an FQL function:

```
TYPE function = RECORD
  CASE kind OF
    defined : (definition : LIST OF function);
    builtin : (opcode : /* internal code */ ;
               environment : LIST OF object)
  END
```

The function APPLY takes two operands -- a functional expression (transitively) mapping $\alpha \rightarrow \beta$ and an object of type α -- and produces, as its value, an object of type β. Specifically:

```
FUNCTION apply (fexpr : LIST OF function ;
                ob : object) : object;
  BEGIN
    REPEAT
      WITH first (fexpr) DO
        CASE kind OF
          defined : ob := apply (definition,ob);
          builtin : ob := subr (opcode,environment,ob)
          END;
      fexpr := rest (fexpr)
    UNTIL null (fexpr);
    apply := ob
  END
```

Each function within the given expression is applied in turn to the "current" object, with the final result serving as

the value of APPLY. If a typical function is not primitive (i.e., is user-defined) then APPLY is invoked recursively on its "definition"; otherwise, the function's "environment" together with the current object are passed by SUBR to the appropriate system routine as designated by the "opcode". We should, at this time, mention that a number of these routines -- those especially concerned with stream-manipulation -- will themselves invoke the function APPLY in a manner similar to PRINT. Indeed, APPLY is called from within the procedure EVAL which, given a reference to a "form", replaces it with the object it denotes:

```
PROCEDURE eval (ob : object);
  BEGIN
    CASE ob.kind OF
      form : ob := apply (ob.fun,ob.arg);
      ELSE :
      END
  END
```

This procedure is invariably invoked by those special system routines implementing FQL operators whose operands are normally tuples (e.g., "+", AND, CONS, etc.). In essence, EVAL is used as a mechanism for transforming a parameter called initially by "name" into one called subsequently by "value".

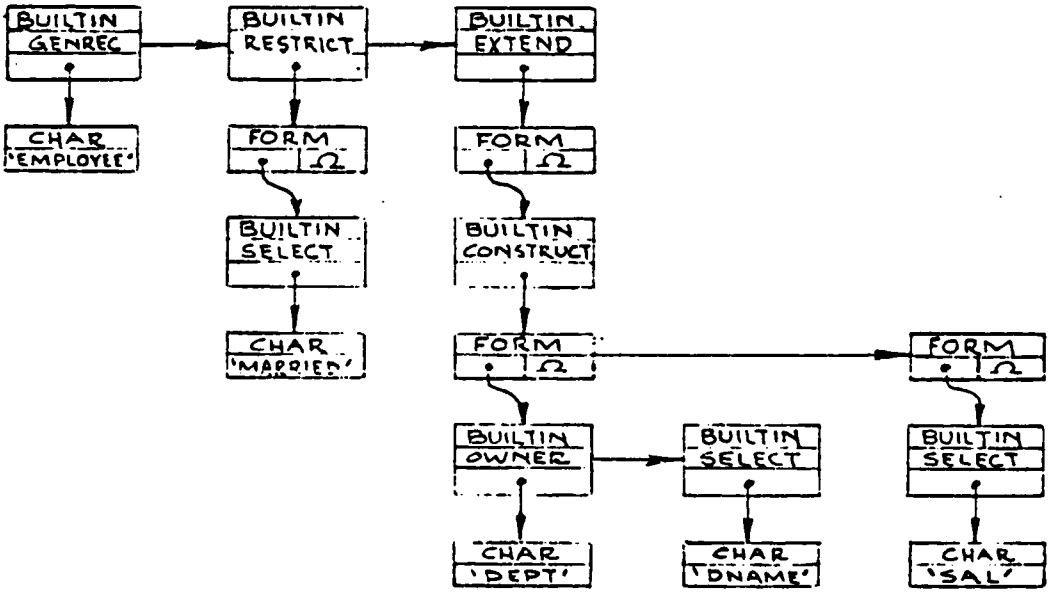
3.3 An Example

To better understand the manner in which the internal representation of functions and objects are manipulated

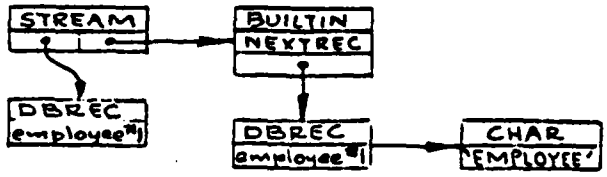
within the FQL processor, we shall selectively trace the execution of the sample query posed at the outset in section 3.1: "the department name and salary of each married employee". Once again, the query may be written as follows:

!EMPLOYEE.|MARRIED.*[DEPT.DNAME,SAL]

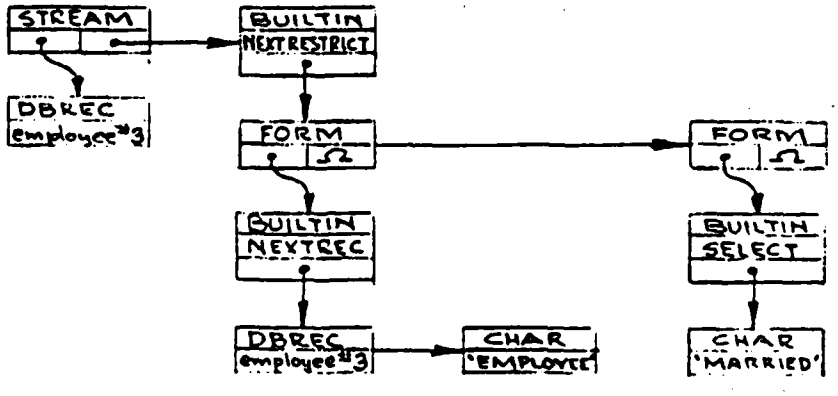
Upon entry of this expression into the system the compiler generates its corresponding internal representation as a list of functions. That is:



Subsequently, the function APPLY iterates through this data-structure, producing as its final value a stream of CHAR-NUM pairs. In the interim, though, several streams of EMPLOYEEs are generated. The first of these results from applying the constant function GENREC (viz., !EMPLOYEE) to some anonymous object, Ω:

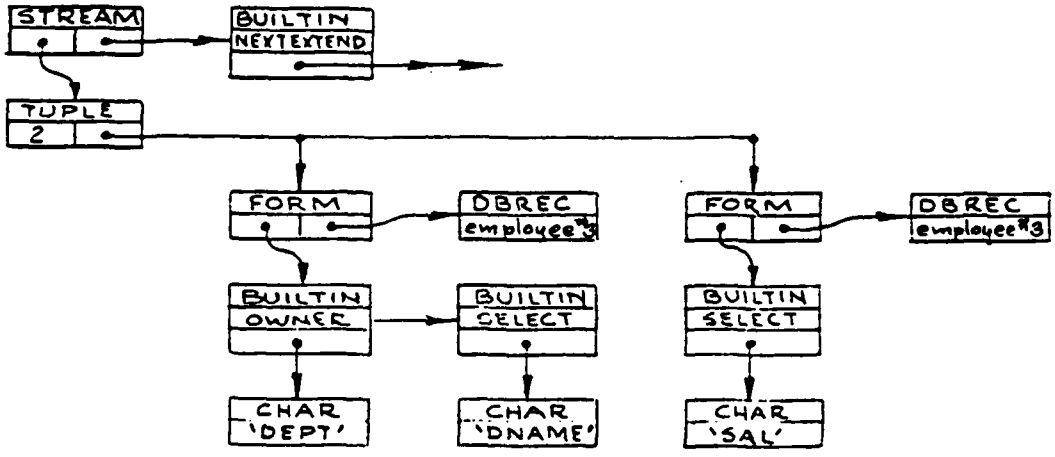


(The head of this stream contains a reference to the first employee in sequence; the tail of this stream is a constant generic function which, when applied with respect to the environment shown, returns a stream of the remaining employees.) This particular object then serves as an operand for RESTRICT (viz., |MARRIED) which, in turn, APPLYs the predicate within its environment to the head of the given stream; and until this predicate tests "true", RESTRICT will consume the stream by continually APPLYing its tail. The stream ultimately produced by RESTRICT may, then, look something like this:



(Here, the tail is represented by the constant function

NEXTRESTRICT whose environment not only includes the original predicate but the tail of yet another stream of EMPLOYEEs.) The final result of the query, that object returned by EXTEND (viz., *[DEPT.DNAME,SAL]), is a rather complex structure whose head is a tuple of forms. In part, the object is depicted as follows:



(The environment of the constant function NEXTEXTEND comprising the tail of this stream would, in turn, not only include the original function being extended but the tail of that stream returned previously by RESTRICT as well.)

As far as output of this object is concerned the procedure is as follows: PRINT receives a stream, prefixes the output of its elements with a "{", and invokes itself recursively on the stream's "head". PRINT now receives a tuple whose components, when EVALuated, produce the department name and salary of the first married employee;

following their respective computation these scalar values are output through further calls upon PRINT and are collectively enclosed in a pair of brackets. The original invocation of PRINT then proceeds to APPLY the tail of the given stream; this activates the internal operator NEXTEXTEND that, in turn, activates the operator NEXTRESTRICT which finally causes NEXTREC to yield a new stream of EMPLOYEEs subject to further restriction and extension. PRINT continues to coerce the elements from the stream's tail in this manner until empty, at which time a "}" is output and the procedure terminates.

* * *

Before leaving the topic of implementation, a few words are in order regarding the issue of storage management. Within the FQL processor the allocation and (especially) the reclamation of store proceeds synchronously with the remainder of the system; which is to say there is no general-purpose garbage collector. To illustrate this discipline of storage management, we shall look briefly at the manner in which PRINT handles streams; indeed, the bulk of the responsibility for storage management in fact resides with this procedure which not only creates objects through invocation of the interpreter but, after output, destroys them as well [Frie 76b]. Consider, then, the following amendations to the definition of PRINT found in section 3.2:

```
        .
        .
stream : BEGIN
  write ('{');
  WHILE not empty (ob) DO BEGIN
    print (share (ob.head));
    obl := apply (ob.tail,Ω);
    free (ob);
    ob := obl
    END;
  free (ob);
  write ('}')
  END
        .
        .
```

After first PRINTing its "head" and then generating a new stream (obl) by APPLYing its "tail", storage used by the current stream (ob) is liberated by the auxiliary procedure FREE; at which point the new stream produced by APPLY becomes the current object. (We should mention that in reality variables such as "ob" and "obl" are only references, thus the assignment of one object to another does not require extensive movement of data.) Incidently, storage is managed in much the same way within APPLY: application of some function within a given expression to a "current" object results in a new object; the former is then FREEd and replaced by the latter.

Another aspect of storage management illustrated by PRINT concerns the explicit sharing of internal data-structures. Since this procedure does as a rule attempt to liberate all storage utilized by its operand, a recursive invocation of PRINT upon the "head" of a given stream will effectively require a copy of this object, lest

the same space will be reclaimed erroneously on multiple occasions. This process of continually copying internal objects and functions can, however, become prohibitively expensive. The alternative adopted within the present implementation involves the sharing of common data; one may, though, view these shared objects and functions as constituting "virtual" copies since the system uses such data in a "read-only" fashion -- that is to say, these structures are never selectively updated. Storage is then managed by means of a reference count maintained for each internal data-structure: this value, unity upon initial allocation, is incremented each time an object or function is SHAREd; when a structure is FREEd, its reference count is decremented and, if zero, storage is reclaimed after (recursively) liberating any other objects or functions it may reference. (Use of this type of storage management scheme is, however, contingent upon the absence of cyclic structures.) Indeed, we should point out that large parts of the data-structures depicted in section 3.3 are shared throughout execution, thus the overhead incurred through repetitive attempts at freeing store is in fact minimal.

4.0 IN PROSPECT

While FQL has proven itself successful in providing a powerful and structured interface to a CODASYL database management system, much work remains to be done with the language. We therefore consider several broad areas for future research and development:

Syntactic, Semantic, and Pragmatic Enhancements

In its present form FQL is somewhat ungainly as an end-user query language. This situation may, however, be alleviated somewhat through a number of enhancements to the system. For one, the specification of input- and output-types within function definitions could be eliminated at the cost of either some run-time checks by the FQL interpreter or else an inference mechanism within the compiler. Besides streamlining the user-interface, the absence of explicit type specifications facilitates definition of generic or polymorphic operators: one should not, for instance, have to redefine the function MEMBER, a mapping from [CHAR,*CHAR] to BOOL (cf. section 2.4), when given, say, numbers instead of character-strings. This kind of generality would indeed be necessary were a facility for defining new functional forms included within the language since, more often than not, the parameters of functionals are arbitrary operators of the form f: $\alpha \rightarrow \beta$.

Other enhancements to FQL might include matters as trivial as allowing infix notation for the standard dyadic operators (e.g., "+") which many may find more convenient. It may also be possible to have the omnipresent "*" functional automatically inserted: much as APL generalizes its scalar functions over vectors and matrices, most scalar functions in FQL have an obvious extension over streams, streams of streams, and so forth. By simplifying matters in this way, the function TC defined in section 2.5 becomes somewhat more readable:

```
TC = COST + !PT.(QTY x COMP.TC)./+
```

On a broader scale, we wish to reiterate FQL's potential role as an intermediate foundation upon which one may build more elaborate user-interfaces, be they oriented toward the first-order predicate calculus or natural language.

A Relational Interface

We have suggested that the conceptual differences between most database management systems can be uniformly subsumed within the functional data-model underlying FQL. One of the obvious extensions to the language is constructing an interface to a relational system. Briefly, each relation constitutes a database-dependent scalar and accordingly defines a collection of functions over that type, one for each subset of the relation's domains. Thus, using conventional relational notation, if

EMPLOYEE: (ENAME,MARRIED,SAL,DNAME)

describes a relation then, within FQL's functional data-model, there exists a data-type EMPLOYEE together with functions such that:

```
EMPLOYEE<ENAME>: EMPLOYEE -> CHAR
EMPLOYEE<MARRIED,SAL>: EMPLOYEE -> [BOOL,NUM]
EMPLOYEE<DNAME>: EMPLOYEE -> CHAR
    etc.
```

Generally, given a relation R and a subset d_1, d_2,...d_k of its domains, there is a function denoted by $R\langle d_1, d_2, \ldots d_k \rangle$ which maps into the data-type $[t_1, t_2, \ldots t_k]$ where t_i is the type of d_i $(1 \leq i \leq k)$. (first normal form guarantees that such data-types will always be printable.) It is then an easy matter to realize both these functions together with their inverse using the operators of the relational algebra or calculus, though the problem of producing <u>efficient</u> relational queries from an FQL expression requires further study.

It is interesting to note that a relational database with added semantics (the Smiths' aggregation model is a good example [Smit 77]) often gives rise to a richer functional representation. The knowledge, for instance, that each value within the domain EMPLOYEE.DNAME serves as a unique key of some other relation (say DEPARTMENT) implies a direct mapping between EMPLOYEE and DEPARTMENT (a "natural" join) and allows inference of schemata not unlike those used

throughout this document.

The Update Problem

While FQL does furnish its user with the means for retrieving information from a database management system, the language at this time contains no general mechanism for updating the content of a particular database; that is to say, FQL may be employed for queries but not for transactions. The problem of update is particularly acute within FQL since, as a rule, a functional programming language will preclude the assignment of variables. One solution is to relax this constraint somewhat and adopt a convention whereby all output from a "top-level" FQL expression must be directed (assigned) to some logical consumer. For instance, a query that returns "the salary of each married employee", given the database of section 2.1, in one sense involves assignment of the value returned by an appropriate FQL expression to the teletype:

```
!EMPLOYEE.|MARRIED.*SAL  =>  TTY ;
```

(Conceptually, it is this explicit coupling of an expression to some consumer that initiates computation of the former's value.)

On the other hand, updating the database may, in the abstract, be viewed as assigning the output of an FQL expression to a particular database-dependent function. Consider, as an example, a transaction which "adds ten

dollars to the salary of all married employees":

```
!EMPLOYEE.|MARRIED.*[ID,[SAL,#10].+]  =>  SAL ;
```

Here, the expression to the left produces, as its value, a "function" compatible with SAL; literally, a stream of [EMPLOYEE,NUM] pairs. When consumed by SAL, this explicit denotation of a mapping may actually be used to direct the update of the database, though in principle it is the function itself that has been altered. Addition and deletion of database-dependent scalars can be similarly accomodated by assignment to functions such as !EMPLOYEE.

A Functional Database Management System

Were FQL used as a common mode of communication within a network of heterogeneous databases, it would become readily apparent that supplementary database management facilites are not only desirable but in certain cases necessary: a query issued by some node within the network against a database at some other node may, due to a lack of computational power at the remote site, require transfer of intermediate results to local storage in order to complete processing. This raises the question of constructing a "functional" database management system -- one whose architecture best realizes the more abstract data-model underlying FQL. Within such a system, the schema of a particular database would be described simply in terms of a collection of functions over various data-types; given this

kind of behavioral specification, the system may then exercise a degree of latitude in choosing an appropriate physical representation. The functional database management system should, moreover, be extensible and permit not only the installation of new functions into a database but the declaration of new data-types as well.

* * *

In retrospect, the most important lesson learned from the design, development, and documentation of FQL has been that functional programming, long an area of theoretical interest, possesses practical value as well. Predicating a discipline of solving problems solely through the composition of functions upon "real-world" situations can be fruitful as well as exciting. It is the hope of this author that FQL will both enlighten and inspire others.

REFERENCES

[Aho 79] Aho, A.V. and J.D. Ullman. "Universality of Data Retrieval Languages," in Sixth Annual ACM Symposium on Principles of Programming Languages, ACM, New York, 1979, 110-20.

[Back 78] Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Comm. ACM, XXI (Aug. 1978), 613-41.

[Burg 75] Burge, W.H. Recursive Programming Techniques, Addison-Wesley, Reading, Mass., 1975.

[Cham 74] Chamberlin, D.D. and R.F. Boyce. "SEQUEL: A Structured English Query Language," in Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access, and Control, ACM, New York, 1974, 249-64.

[Cham 76] Chamberlin, D.D. "Relational Data-base Management Systems," ACM Computing Surveys, VIII (March 1976), 43-66.

[Coda 71] CODASYL Data Base Task Group, April 1971 Report, ACM, New York, 1971.

[Codd 70] Codd, E.F. "A Relational Model of Data for Large Shared Data Bases," Comm. ACM, XIII (June 1970), 377-97.

[Codd 71] Codd, E.F. "A Data Base Sublanguage Founded on the Relational Calculus," in Proc. 1971 ACM-SIGFIDET Workshop on Data Design, Access, and Control, ACM, New York, 1971, 35-68.

[Date 77] Date, C.J. An Introduction to Database Systems, Addison-Wesley, Reading, Mass., 1977.

[Frie 76a] Friedman, D.P. and D.S. Wise. "CONS Should Not Evaluate its Arguments," in Automata, Languages, and Programming, S. Michaelson and R. Milner (Eds.), Edinburgh Univ. Press, Edinburgh, 1976, 257-84.

[Frie 76b] Friedman, D.P. and D.S. Wise. "Output Driven Interpretation of Recursive Programs, or Writing Creates and Destroys Data Structure," Information Processing Letters, V (Dec. 1976), 153-60.

[Frie 77] Friedman, D.P. and D.S. Wise. "Aspects of Applicative Programming for File Systems," SIGPLAN Notices, XII (March 1977), 41-55.

[Gerr 78] Gerritsen, R. SEED Reference Manual, International Database Systems, Philadelphia, 1978.

[Land 65] Landin, P.J. "A Correspondence between ALGOL 60 and Church's Lambda Notation," Comm. ACM, VIII (Aug. 1965), 89-101.

[Mart 75] Martin, J. Computer Database Organization, Prentice-Hall, Englewood Cliffs, N.J., 1975.

[Ship 79] Shipman, D.W. "The Functional Data Model and the Data Language DAPLEX," in Proc. of ACM-SIGMOD Internatl. Conf. on Management of Data, forthcoming.

[Smit 77] Smith, J.M. and D.C.P. Smith. "Aggregation and Generalization," ACM Transactions on Database Systems, II (June 1977).

[Ston 76] Stonebraker, M. et. al. "The Design and Implementation of INGRES," ACM Transactions on Database Systems, I (Sept. 1976), 189-222.

[Tayl 76] Taylor, R.W. and R.L. Frank. "CODASYL Data-base Management Systems," ACM Computing Surveys, VIII (March 1976), 67-103.

[Zloo 75] Zloof, M.M. "Query by Example: the Invocation and Definition of Tables and Forms," in Proc. Internatl. Conf. on Very Large Data Bases, ACM, New York, 1975, 1-24.

[Frie 77] Friedman, D.P. and D.S. Wise. "Aspects of Applicative Programming for File Systems," SIGPLAN Notices, XII (March 1977), 41-55.

[Gerr 78] Gerritsen, R. SEED Reference Manual, International Database Systems, Philadelphia, 1978.

[Land 65] Landin, P.J. "A Correspondence between ALGOL 60 and Church's Lambda Notation," Comm. ACM, VIII (Aug. 1965), 89-101.

[Mart 75] Martin, J. Computer Database Organization, Prentice-Hall, Englewood Cliffs, N.J., 1975.

[Ship 79] Shipman, D.W. "The Functional Data Model and the Data Language DAPLEX," in Proc. of ACM-SIGMOD Internatl. Conf. on Management of Data, forthcoming.

[Smit 77] Smith, J.M. and D.C.P. Smith. "Aggregation and Generalization," ACM Transactions on Database Systems, II (June 1977).

[Ston 76] Stonebraker, M. et. al. "The Design and Implementation of INGRES," ACM Transactions on Database Systems, I (Sept. 1976), 189-222.

[Tayl 76] Taylor, R.W. and R.L. Frank. "CODASYL Data-base Management Systems," ACM Computing Surveys, VIII (March 1976), 67-103.

[Zloo 75] Zloof, M.M. "Query by Example: the Invocation and Definition of Tables and Forms," in Proc. Internatl. Conf. on Very Large Data Bases, ACM, New York, 1975, 1-24.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation Center
(12 copies)
Cameron Station
Alexandria, VA 22314

Office of Naval Research
(2 copies)
Information Systems Program
Code 437
Arlington, VA 22217

Office of Naval Research
Code 102IP
Arlington, Virginia 22217

Office of Naval Research
Branch Office
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

New York Area Office

715 Broadway - 5th Floor
New York, NY 10003

Naval Research Laboratory
(6 copies)
Technical Information Division
Code 2627
Washington, DC 20375

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, DC 20380

Office of Naval Research
Code 455
Arlington, VA 22217

Office of Naval Research
Code 458
Arlington, VA 22217

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation & Mathematics Dept.
Bethesda, MD 20084

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, DC 20350

Mr. Kim B. Thompson
Technical Director
Information Systems Division
(OP-911G)
Office of Chief of Naval Operations
Washington, DC 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, NJ 08903
Attn. Dr. Henry Voos

Professor Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, NY 10027

Defense Mapping Agency
Topographic Center
ATTN: Advanced Technology
Division
Code 41300 (Mr. W. Mullison)
6500 Brookes Lane
Washington, D.C. 20315

Commander, Naval Sea Systems Command
Department of the Navy
Washington, D.C. 20362
ATTENTION: (PMS30611)

Major J.P. Pennell
Headquarters, Marine Corps
Washington, D.C. 20380
ATTENTION: Code CCA-40

Captain Richard L. Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York 09501

Professor Mike Athans
Massachusetts Institute of Technology
Dept. of Electrical Engineering and
Computer Science
77 Mass. Avenue
Cambridge, MA 02139