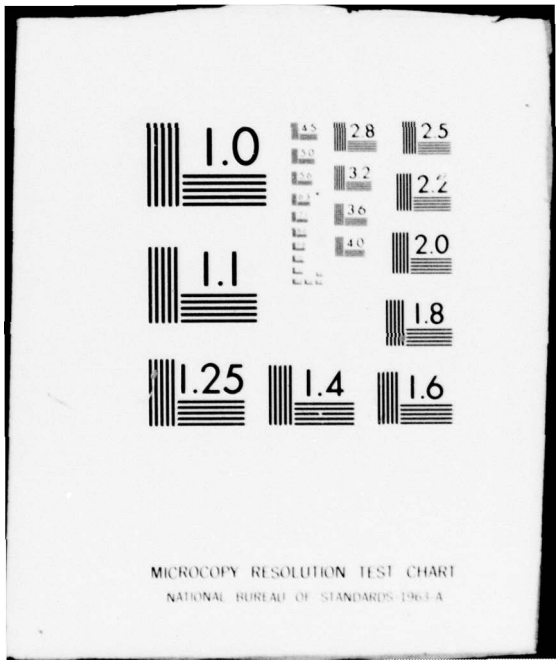
AD-A066 992 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
FQL A FUNCTIONAL QUERY LANGUAGE.(U)
MAR 79 O P BUNEMAN, R E FRANKEL N00014-75-C-0462
UNCLASSIFIED 79-03-05 NL

END
DATE FILMED
6-79
DDC


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
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Wharton

Department of Decision Sciences

(12)

LEVEL II

University of
Pennsylvania
Philadelphia PA 19104

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) 79-03-05	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): (6) FQL — A Functional Query Language	5. TYPE OF REPORT & PERIOD COVERED: (9) Technical Report. Apr 1978 – Mar 1979
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): (10) O. Peter Buneman, Robert E. Frankel	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Decision Sciences, University of Pennsylvania, Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (15) Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research	12. REPORT DATE: (11) Mar 79
	13. NUMBER OF PAGES: (12) 28p. 23
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)

Distribution unlimited

18. SUPPLEMENTARY NOTES

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

applicative languages, query languages, databases, data base models

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

An applicative language based upon recent ideas by John Backus has been developed. The language provides a powerful formalism for the expression of complex database queries. Though currently implemented with an interface to a CODASYL system, the language employs a sufficiently general data model that use with other database management systems is possible. This paper describes the language through a number of examples and outlines its implementation.

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)

FQL -- A Functional Query Language

(preliminary version)

Peter Buneman
Robert E. Frankel

Abstract

An applicative language based upon recent ideas by John Backus has been developed. The language provides a powerful formalism for the expression of complex database queries. Though currently implemented with an interface to a CODASYL system, the language employs a sufficiently general data model that use with other database management systems is possible. This paper describes the language through a number of examples and outlines its implementation.

Authors' address: Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, Pa. 19104

FQL -- A Functional Query Language

1.0 INTRODUCTION

Formal database query languages are extraordinarily varied, ranging from the algebraically based relational languages [3, 10] to those languages provided with some CODASYL systems [4] which give the user direct access to the data-manipulation routines within the database management system. The Functional Query Language described here, FQL, was originally designed and implemented in an effort to provide a powerful and structured interface to a CODASYL database management system. Constructing an interface to other database systems should, however, present little difficulty as the data model used by the language is quite general: FQL may in fact serve as a common query language for communication with different database systems.

FQL is an applicative language; and embodies many of the ideas concerning functional programming systems recently described by Backus [1]. The only control structure in fact available to the user is the abilitiy to combine functions. The means for explicit data reference (i.e., variables) have been purposely omitted from the language. As a result, FQL differs from other query languages in several important respects.

1. There is no notion of data currency: many query systems (the relational languages are a notable exception) operate on a "record-at-a-time" basis

strongly reminiscent of what Backus terms the "von-Neumann bottleneck".

2. Complex queries may be developed incrementally from simpler queries: a query in FQL is no more than another function over the database which, using the mechanisms provided by the language, can be combined with other queries.

3. Full computational power is provided: many query languages lack the ability to do basic arithmetic, let alone recursion (some relational language are particularly weak in this respect).

4. The language itself is independent of any database system: the data model it employs can be interfaced with database systems other than CODASYL.

It should be emphasized that we do not see FQL as the "ideal" end-user query language; nor do we believe that such a language exists. Rather, it is presented as a precise and powerful formalism for the expression of database queries. Our hope is that FQL (or some syntactic variant of it) can serve both as a tool for those wishing to construct complex queries and as an intermediate language into which one's "favorite" query language may be readily translated (an early version of FQL is currently in use as a database interface for a natural language system [7]).

The main purpose of this paper is to provide an informal introduction to FQL. In the following section the rudiments of the language together with simple examples of their application are presented. Some issues of implementation are then outlined. The final section of the paper is devoted to a discussion of the future development of FQL including its extension to other database systems and

its more general role as an applications language.

2.0 THE LANGUAGE

2.1 A Functional View Of Databases

Since FQL is a functional language, we need to adopt a functional view of databases. We regard a database as a collection of functions over various data-types. Figure 1 shows the schema of a (very simple) database containing entities of type EMPLOYEE and of type DEPARTMENT.

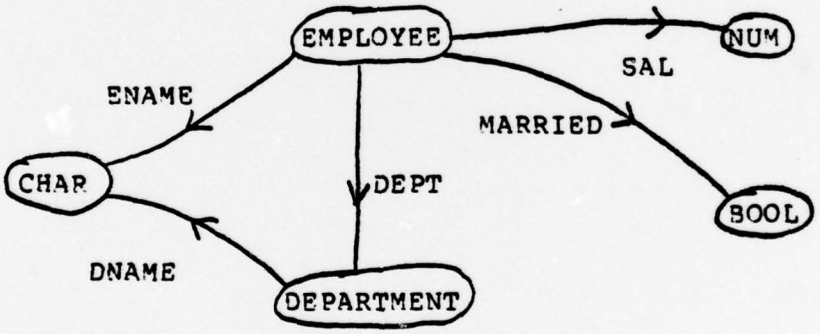


Figure 1

The function DEPT in this example represents a mapping between these types; that is, given an EMPLOYEE, DEPT returns that DEPARTMENT in which he works. In addition, this database furnishes functions which map these entities into 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. To summarize using conventional notation:

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

Of the five data-types seen here, CHAR, NUM, and BOOL are standard in that they exist independently of any database. On the other hand, the types EMPLOYEE and DEPARTMENT together with the five functions described above are specific to this database. It should also be noted that we regard all of these types as scalar. Information about EMPLOYEEs or DEPARTMENTs may only be obtained through those database functions which map these entities into "printable" types (e.g., CHAR).

2.2 Creating New Functions

Knowing that the database defines a set of functions we need mechanisms -- functional forms as Backus terms them -- for combining these operators to create new and more powerful functions. Of these mechanisms, the most fundamental is composition. One might, for instance, wish to define a function which, given an employee, returns the name of the department in which he works:

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

This example shows the syntax of an FQL function definition: DEPTNAME is declared to be a function from EMPLOYEE to

CHAR(acter string) and defined to be the composition of the functions DEPT and DNAME (as denoted by the operator "."). Observe that these functions are composed, and hence evaluated, from left to right (reverse Polish): the DEPARTMENT returned by applying DEPT to an EMPLOYEE serves as the operand for DNAME which, in turn, produces a value of type CHAR. Using reverse Polish for functional composition in fact seems quite natural: the left to right order of the functions determines a corresponding path through the database schema.

So far, we have considered only those functions within the database which map scalars into other scalars; and thus any functions one might define using composition would be scalar as well. We have yet to deal with collections of objects such as the set of all EMPLOYEEs within a particular DEPARTMENT. To do so, we must augment our view of the database to include the inverse of functions. Returning to our sample database the inverse of the function DEPT, written !DEPT, maps a DEPARTMENT into a sequence of all those EMPLOYEEs who belong to the given department; similarly, the function !DNAME maps a CHAR(acter string) into a sequence of DEPARTMENTs[1]. We shall use the term stream to refer to such a sequence of objects of a given data-type. (A stream, following Landin [8] and Burge [2],

---

1. Strictly speaking !DNAME is not the mathematical inverse of DNAME as the inverse of a function contains no notion of sequentiality.

is a "virtual" sequence of objects whose physical representation should be of no concern to the programmer -- it may be a list in primary store, a file in secondary store, a generating function, or some combination of these.) Whether or not the inverse of a given function is available depends 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 mechanisims for representing inverses of functions when they are required. For example, !DEPT would be implemented through a CODASYL set, while !DNAME might be implemented using a hash table.

We return now to the task of creating new functions, in particular functions which map streams into other streams. For this, we introduce two additional functional forms, extension and restriction. The first of these allows, say, the function SAL (a mapping from EMPLOYEE to NUM) to be "extended" into a function, denoted *SAL, which, given a stream of EMPLOYEEs, returns a stream of NUMs by applying the function SAL to each EMPLOYEE within the stream. The second functional form allows streams to be filtered by predicates over individual elements: the function |MARRIED maps a stream of EMPLOYEEs into the sub-stream of EMPLOYEEs satisfying the condition that they be MARRIED. Note that while extension will preserve the length of a given stream, restriction generally will return fewer elements. As an example, these operators are used to create a function which

returns a stream of salaries of all married employees within a given department:

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

Here, MARRIED-SALS is defined as a function from type DEPARTMENT into a stream of NUMs (we use *NUM to denote the data-type of a stream of entities of type NUM). It is formed by applying the inverse of DEPT, !DEPT, to produce a stream of EMPLOYEEs; this stream is restricted to that sub-stream satisfying the predicate MARRIED; the function SAL is then applied to each of the remaining EMPLOYEEs yielding a stream of NUMs.

A final functional form, construction, is needed to create functions that returns tuples of objects; for instance, an employee's name and salary. In that case the notation [ENAME,SAL] signifies a mapping from EMPLOYEE to a pair comprising a CHAR and a NUM. Thus we would write:

```
NAME-AND-SALS: EMPLOYEE->[CHAR,NUM] = [ENAME,SAL];
```

The range of this function, [CHAR,NUM], denotes the data-type of tuples of type CHAR and NUM. Tuples, in fact, will become important when operators such as addition are introduced since, by design, all FQL functions (primitive or composite) are monadic: the operator "+" represents a mapping from a pair of NUMs into a NUM; e.g., + : [NUM,NUM] -> NUM.

2.3 Queries

A query is a special kind of function whose range is some "printable" object. (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 an example, we might wish to know "the department names and salaries of all married employees." To realize this query, however, we need access to the stream of all employees within the database. Again, our functional view of the database must be extended, this time to include a set of "constant" functions. In our database these include:

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

(The absence of a data type to the left of the "->" indicates a constant function.) The functions !EMPLOYEE and !DEPARTMENT respectively return streams of all values of type EMPLOYEE and DEPARTMENT currently in the database. Of course, these functions are not truly constants in that they are database dependent and that their values can, and often do, change over time. Returning, then, to our query that produces the department name and salary of each married employee:

```
Q1: ->*[CHAR,NUM] = !EMPLOYEE.|MARRIED.*[DEPT.DNAME,SAL];
```

For convenience, we will assume that the database is current and thus Q1, like the function !EMPLOYEE, is a "constant"

whose domain will remain unspecified. Also observe that the function being extended by the "*" in this example is itself the result of applying the construction operator to a pair of functions, one of which is the composition of yet two other functions.

To recapitulate, we have viewed a database as a collection of functions over various data-types and have presented four functionals -- composition, extension, restriction, construction -- for combining these functions into new functions; and ultimately into queries. We have also introduced two modes for structuring types -- streams (*α) and tuples ([α,β...]) -- where Greek letters denote arbitrary data-types. The types of the functions FQL's functionals produce are summarized in the following where lower-case letters signify functions.

1. Composition. If f and g are such that f: α ->β and g: β ->γ then f.g: α ->γ.
2. Extension. If f: α ->β then *f operates upon a stream of these types; i.e., *f: *α->*β.
3. Restriction. If p is a predicate over α (i.e., p: α ->bool) then |p: *α->*α.
4. Construction. If $f_1 : \alpha \rightarrow \beta_1$, $f_2 : \alpha \rightarrow \beta_2 \ldots$ $f_n : \alpha \rightarrow \beta_n$ then $[f_1, f_2 \ldots f_n] : \alpha \rightarrow [\beta_1, \beta_2 \ldots \beta_n]$.

2.4 Standard Functions

The class of queries one can formulate using only the functions given by the database is rather limited. Our

language therefore contains an array of standard functions including the familiar arithmetic, relational, and boolean operators, a set of constructors and selectors for structured data-types, and a number of "stream-reducing" functions which map streams into scalars. To point out the use of several of these functions (a comprehensive listing may be found in an appendix) consider a query which returns "the names of those employees who earn above the average salary for their department. First, though, let us define a function AVRG which computes the mean of stream of numbers:

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

Here, the function "/+" (borrowing from APL) sums the elements of the given stream while "LEN" returns its length; this pair of NUM(bers) then serves as the operand for the division ("/") function. Next, let us define a predicate over EMPLOYEEs:

```
EARNSMORE: EMPLOYEE->BOOL =
           [SAL,DEPT.!DEPT.*SAL.AVRG].GT;
```

EARNSMORE returns the value "true" if the given EMPLOYEE earns more than the average salary among his co-workers and "false" otherwise. The sub-expression "DEPT.!DEPT" returns the co-workers per se; *SAL retrieves their respective wages, and AVRG computes the mean; this value is then compared to the original EMPLOYEE's salary using the relational operator "GT." And finally, the query:

```
Q2: ->*CHAR = !EMPLOYEE.|EARNSMORE.*ENAME;
```

As a further demonstration of the language we now turn to a more complex query: "the names of those departments and all of their employees in which the average salary is below $20,000 and some of the employees are not married." First, the query itself:

```
Q3: ->*[CHAR,*CHAR] =
    !DEPARTMENT.|([P1,P2].AND).*[DNAME,!DEPT.*ENAME];
```

The query is formed through restricting the stream of all DEPARTMENTs by the conjunction of predicates P1 and P2 as specified below (note the use of parentheses to enforce the scope of the restriction operator); for each DEPARTMENT in this sub-stream a pair consisting of its name and a stream of its EMPLOYEE's names is then constructed. The predicates P1 and P2 are defined as follows:

```
P1: DEPARTMENT->BOOL = [!DEPT.*SAL.AVRG,#20000].LT;
P2: DEPARTMENT->BOOL = !DEPT.*(MARRIED.NOT)./OR;
```

The first predicate compares the average salary earned in a given DEPARTMENT with the value 20,000 (more precisely, with the value of the constant function 20000). The second predicate tests for the presence of some EMPLOYEE who is not married: the expression *(MARRIED.NOT) yields a stream of BOOL(eans) while the function "/OR" returns the value "true" if some member of this stream is "true."

2.5 A Bill-of-Materials Processor

As a final example, we will attack the infamous bill-of-materials problem which, to our 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-up out of other parts; and so on. The specific task we shall address 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 may require four piston assemblies and two carburetors. Figure 2 depicts a possible schema for such a database:

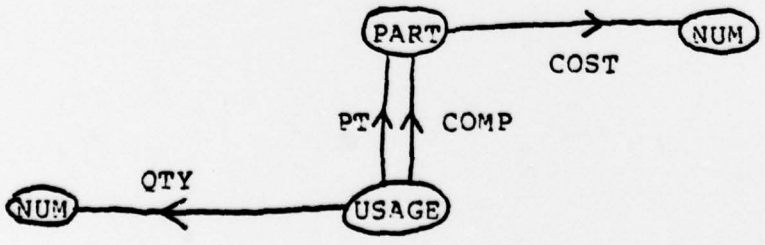


Figure 2

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 function TC which computes a part's total cost:

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

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.

What is remarkable about this particular function is that its definition, despite recursion, includes no explicit basis for termination! (FQL queries do not usually require the IF-THEN-ELSE construct normally associated with termination in recursive functions.) 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.

Finally, we should mention that FQL does include a set of basic stream-manipulating primitives based upon the list-processing functions, CONS, CAR, CDR, etc. of LISP. (These are described in an appendix.) It is interesting to note that for most database queries explicit use of these operators is not required: they are implicit in extension and restriction. Constructing a bill-of-materials processor which lists all the sub-parts of a given part would, however, call for a direct use of CONS. (Again, this is left as an exercise for the reader.)

3.0 IMPLEMENTATION

In its current form, FQL is implemented in PASCAL as an interface to SEED [6], a CODASYL-based database management system written in FORTRAN, and is running on The Wharton School's DEC-10. Little difficulty is foreseen in transferring FQL to other machines or, if necessary, re-writing its source code in some other programming language. Interfacing to other CODASYL systems may require some additional code: unless routines are provided for run-time interrogation of the database schema, as they are in SEED, a pre-processor would be necessary to translate

from the CODASYL data-definition language (DDL) into a separate representation of the schema used by FQL.

The mapping between a CODASYL schema and the functional model presented above proves quite straightforward[2]. Roughly speaking, sets and items correspond to functions while the records become types. The "functional" database depicted earlier in Figure 1 would, for instance, derive from a schema in which DEPARTMENT and EMPLOYEE were records, DNAME and ENAME were respectively items within these records of type CHAR, and DEPT were a set owned by DEPARTMENT and populated by EMPLOYEEs. Note, though, that the set DEPT actually corresponds to the inverse function !DEPT; the function DEPT itself takes a member-record into its owner. It should also be mentioned that inverse functions such as !DNAME are only available when the corresponding item (DNAME in this case) serves as a CALC key.

Strictly speaking, CODASYL sets correspond to functions only when membership is both MANDATORY and AUTOMATIC. In other cases we may only assume that a set defines a partial function. In order to cope with partial functions, a standard object, UNDEF, has been introduced as a member of

---

2. Some CODASYL constructs cannot be represented in FQL's data model at present. These include arrays together with sets in which more than one record type may be owned. There appear to be no fundamental difficulties in extending FQL to cope with these.

every type; and a standard predicate, DEFINED, is available to test whether or not an object is defined. Partial functions can then be represented by (total) functions which may, at times, return UNDEF.

One of the more interesting facets of the implementation of FQL is the manner in which streams are handled internally. The problem here is to support the user's illusion of constructing and traversing (possibly very long) lists of data without monopolizing large amounts of primary store. To some extent out solution follows from the work of Friedman and Wise [5] though there are significant differences. To consider an example, the "constant" function !EMPLOYEE does not literally return a list of all EMPLOYEEs currently in the database (to do so would not only be impractical but in certain cases impossible). Rather, this function generates a stream represented by an ordered-pair comprising its head, the first EMPLOYEE in sequence, and its tail, another function which, when applied, produces a stream of the remaining EMPLOYEEs (i.e., it produces another ordered-pair). Among the advantages of this scheme is that the amount of primary store required to process a sequence of indeterminate length remains constant.

4.0 DISCUSSION

In this section we shall briefly discuss extensions to FQL and further related research areas. The present syntax of FQL makes it somewhat ungainly as an end-user query language. There are a number of changes that could be made to alleviate this situation. At the cost of some run-time checking, the type declarations on the left hand side of function definitions could be eliminated. Also, infix representation of the dyadic arithmetic and relational operators may be found more convenient. It is also possible to have the "*" functional automatically inserted: much as APL generalizes its standard functions over arrays, most functions in FQL have an obvious extension over streams. Using this simplified syntax, the bill-of-materials query described above reduces to:

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

which is a somewhat more readable version of the function.

We have suggested that the functional schema used by FQL is general enough to allow representation of other database systems. One of the obvious extensions to FQL is an interface to a relational system. Briefly, each relation defines a data-type and a set of functions, one for each subset of its domains. Thus, using conventional relational database notation, if

EMP: (ENAME, DEPT#, SAL)

describes a relation then, in the functional description, there is a data type EMP together with functions such as:

```
EMP<ENAME, SAL>: EMP ->[CHAR,NUM]
EMP<DEPT#>: EMP->NUM
    etc.
```

Generally, given a relation R, and a subset $d_1, d_2 \ldots d_K$ of its domains, there is a function denoted by $R<d_1, d_2 \ldots d_K>$ which maps into the data-type $[t_1, t_2 \ldots t_K]$ where t_i is the data type of d_i $(1<i<k)$. First normal form guarantees that such data types are always printable. It is an easy matter to implement these functions and their inverses using the operators of the relational calculus. It may, however, be possible to translate FQL queries more directly into a relational query language, though the problem of producing efficient relational queries from an FQL definition requires further work.

It is interesting to note that a relational database with added semantics (the Smiths' aggregation model [9] is a good example) often gives rise to a much simpler functional representation. Direct functions between relations (the "natural" joins) are available and schemata not unlike those used in this paper may be directly inferred from the semantics.

We would also like to extend FQL to be a more general database applications language. The problem here, as in all applicative languages, is that of update. It should be possible to add the ability to update functions; and if this were done, it would give the user the ability to define high level transactions in a structured fashion. This may well be of advantage when working in a shared environment where many update anomalies are caused by the user having explicit control over data currency. Other additions we would like to see to FQL include functions which describe the type of a database entity and what functions are available. This would allow queries to interrogate the structure of the functional schema and greatly enhance FQL's use as a general-purpose applications language.

5.0 REFERENCES

1. Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Comm. ACM, 21, 613-641.

2. Burge, W.H. Recursive Programming Techniques. Addison-Wesley, Reading, Mass., 1975.

3. Chamberlin, D.D. and R.F. Boyce. SEQUEL: A Structured English Query Language. Proc. ACM SIGMOD Workshop, 1974.

4. Data Base Task Group April 1971 Report. ACM, New York 1971.

5. Friedman, D.P. and Wise, D.S. CONS should not evaluate its arguments. In Automata, Languages, and Programming. Edinburgh Univ. Press, Edinburgh 1976.

6. Gerritsen, R. Seed Reference Manual. International Database Systems, Philadelphia (1978).

7. Kaplan, S.J. CO-OP: A Natural Language Co-operative Query System. PhD. Dissertation, Moore School, University of Pennsylvania.

8. Landin, P.J. A Correspondence between ALGOL 60 and Church's Lambda Notation. Comm. ACM, 8, 89-101.

9. Smith, J.M and D.C.P. Smith. Aggregation and Generalization. ACM Transactions on Database Systems 2 (June 1977).

10. Stonebraker, M. et al. The Design and Implementation of INGRES. ACM Tranasactions on Database Systems, Sept. 1976.

Appendix 1 -- FQL Syntax

Below is given the syntax for a function defintion, a data-type, a functional expression, and a function itself. Optional components are denoted by "{"..."}^{1}" while "{"..."}$^{+}$" signifies a set of elements may occur an arbitrary number of times.

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

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

<fexpr> ::= <function>{,<function>}+

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

Appendix 2 -- Standard Functions

The standard functions supported by FQL are grouped here by category. Where appropriate, additional explanation is provided.

Arithmetic functions

The functions +, -, x, /, and MOD all map from [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 Functions

The operators EQ, NE, GT, LT, GE, and LE map from either [NUM,NUM] or from [CHAR,CHAR] into BOOL. The functions AND and OR return a BOOL given [BOOL,BOOL]; the complement NOT takes a 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.

Constant Functions

The notation #<number> represents a mapping ->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 Functions

Given a non-empty stream, the operation HD returns its first element (*α->α) while the operation TL returns a stream of the remaining elements (*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 Functions

The function LEN computes the length of a given stream and is thus a mapping from *α into NUM. CONC maps 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 "distributed" over the stream of α's.

Miscellaneous Functions

The function i (i=1,2...n) selects a component from a tuple; i.e., $[\alpha_1, \alpha_2 \ldots \alpha_n] \rightarrow \alpha_i$. Finally, ID represents the identity mapping $\alpha \rightarrow \alpha$.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation
Cameron Station
Alexandria, VA 22314
12 copies

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

Office of Naval Research
New York Area Office
715 Broadway - 5th Floor
New York, N.Y. 10003

Dr. A.L. Slafkosky
Scientific Advisor (RD-1)
Commandant of the Marine Corps
Washington D.C. 20380

Office of Naval Research
Code 458
Arlington, VA 22217

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

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

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

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 copies

Office of Naval Research
Code 455
Arlington, VA 2217

Naval Electronics Lab. Cener
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center

Computation and Mathematic Dept.
Bethesda, MD 20084

Mr. Kim Thompson
Technical director
Information Systems Division
OP-911G
Office of Chief Naval Operations
Washington, D.C. 20350

Prof. Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, N.Y. 10027

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

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

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
OP-916D
Office of Chief of Naval Research
Washington, D.C. 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, N.J. 08903
ATTENTION: Dr. Henry Voos

Defense Mapping Agency
Topographic Center
ATTN: Advanced Technology Div.
Code 41300
6500 Brookes Lane
Washington, D.C. 20315

(12)

FQL -- A FUNCTIONAL QUERY LANGUAGE

O. Peter Buneman
Robert E. Frankel

79-03-05

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

Research supported in part by the Office of Naval Research under Contract N00014-75-C-0462.

-408757