# PROGRAMMING LANGUAGES FOR RELATIONAL DATA BASE SYSTEMS

by

C. J. Prenner and L. A. Rowe

Memorandum No. UCB/ERL M78/54

27 July 1978

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# Programming languages for relational data base systems*

*by* CHARLES J. PRENNER and LAWRENCE A. ROWE

*University of California at Berkeley*
Berkeley, California

## INTRODUCTION

There has been a remarkable growth in the use of data base management systems over the past decade. This has resulted from the recognition that many practical applications, previously implemented by special purpose programs, can be developed more easily through the use of a general data base facility. Here, we will only be concerned with relational data base systems. The advantages of such systems have been discussed at length in the literature.[7,9,10]

In modern data base management systems, users typically interact with data bases via a query language.[4,6,8,20,41] These languages, which are often interactive, allow creation and deletion of data bases as well as retrieval from, and updating of, existing data bases. Because query languages have not provided a complete programming environment, most of these languages have been coupled to existing programming languages. Application programs consist of statements in the programming language intermixed with statements in the query language to access the data base system when required.

Previously investigated approaches to providing such access include the definition of subroutines to execute data base functions and the embedding of data base constructs into an existing language, using a preprocessor to translate these constructs into run-time calls on a data base system (e.g., EQUEL,[2] and the embedding of SEQUEL in PL/I[6]). Because the primary focus of this work was to provide data base access from a programming language, little attention was paid to the programming environment resulting from the combination of the two languages.

While a given programming language and a given query language may each be satisfactory in isolation, their combination may be less than satisfactory because neither one was designed for this more general setting. In this paper we discuss techniques for improving the programming environment for data base applications. Our hypothesis is that a superior environment can be realized by incorporating the data base operations as part of the programming language itself. In the second section we outline some limitations associated with previously proposed environments. These stem primarily from the fact that the programming language is totally unaware of the data base. In the third section we illustrate some of the benefits provided by an integrated approach through the application of current techniques from programming language research.

* This research was supported in part by U.S. Army Research Office Grant DAAG 29-76-G-0245, and by the University of California under a Faculty Research Development Grant.

## CURRENT APPROACHES

Most current approaches to providing data base access from a programming language are constrained by the fact that the language system has no knowledge about the data base. We divide our discussion of the impact of this constraint into five areas: data base interface, data base operations, type system, compilation, and abstraction.

### *Data base interface*

As mentioned previously, the two primary approaches to providing data base access have been to provide subroutines for data base functions and to embed data base constructs through the use of a preprocessor.

The subroutine approach suffers from the limitations inherent in any use of subroutines to extend the semantic space of a language; programs become essentially a series of calls which are often unreadable because of the many (bookkeeping) arguments which must be passed to the subroutine. No syntactic aids are provided to make the use of these arguments understandable.

The preprocessor approach is somewhat better in that an actual query language is used. Program statements and query language statements can be freely intermixed. In most systems the preprocessor does not have to parse the entire language because query language statements are tagged with some distinguished symbol. The preprocessor removes the query language statements, replacing them with calls upon the data base subsystem, before the program is passed to the language translator. The programmer transacts with a readable version of the program and only the language translator need be concerned with the less readable version.

Unfortunately, the preprocessor approach is limited as well. Since the query language is not part of the programming language, communication between the programming language and the data base system can become awkward.

For example, it is often necessary to use implicitly defined global variables to communicate the status of queries. In some contexts, program variables which are to be made accessible to the data base system must be preceded by "special symbols" so that they may be recognized by the preprocessor. In some preprocessors arbitrary constraints are imposed as to which arguments to a data base operation can be variables and which must be constants. In other implementations, even though the language being extended is strongly typed, the same level of checking is not extended to the data base objects and their operations. Detailed examples illustrating these problems are given in a previous paper.[35]

Although preprocessing the entire language would remove some of these limitations, this approach is still constrained by the fact that (practically speaking) no amount of preprocessing can extend the language to incorporate relations with the same amount of symmetry, type checking and protection available for built-in objects.

*Data base operations*

In considering the query language as part of the programming language one finds that the notation and orientation of the two are quite different. Commonly used programming languages are procedural, with built-in facilities for iteration, manipulation of complex data structures, and procedural abstraction. On the other hand, most query languages are non-procedural. In the context of a procedurally oriented programming language these non-procedural operations seem out of place. To provide a consistent programming environment the objects manipulated by the data base system and their attendant operations and control structures should be related to the objects, operations and control structures of the language itself. For example, a join operation may be thought of as a nested iteration over relations yet the operation "join" usually appears in a form completely unrelated to the iteration operations in the programming language. We do not advocate that higher level operators such as joins be excluded from a language. Instead, we suggest that they be defined as extensions to the language in terms of existing primitives.

*Type system*

Because the programming language has no knowledge of the data base system, the type systems of the two are usually distinct. Data base objects cannot be manipulated in the programming language with the same ease and flexibility as language objects. For example, records, as used in a relational data base system, often are not treated as composite structures in the language environment (e.g., EQUEL or SEQUEL). This means that values must be copied individually from data base records into program variables before the values can be used. Thus, a record from a data base cannot be an argument to a procedure nor an operand to an otherwise meaningful operation (e.g., record selection or the right-hand-side of an assignment). The same situation holds for relations. For example, relations may not be passed as parameters to functions. In addition, it is impossible to operate on relation types, say, to obtain the count of the number of columns in the relation, or to obtain the domain of a column as a type. The existence of two type systems may require conversions to be performed on objects as they flow between the two systems. One type system controlling data access whether in main memory or secondary storage would be more efficient because it removes the need for such conversions.

*Compilation*

In programming languages, there is often a tradeoff between program flexibility and run-time efficiency. Most data base systems are committed to one extreme or the other depending upon whether data base operations are interpreted (less efficient but more flexible,* e.g., INGRES) or compiled (more efficient but less flexible, e.g., SEQUEL). When a language is committed to one extreme the resulting design decisions make the language awkward to use, or less efficient, when the application requirements are not matched to that commitment. For example, consider the treatment of relation names in SEQUEL. In order for a query to appear explicitly in the text, all relation names must be constant (to allow compilation to be performed). If dynamically varying names are desired, then a string for the query text must be constructed and the relation name inserted into it at run-time. On the other hand, INGRES allows dynamically varying relation names but cannot utilize the fact that a relation name is constant in a given instance. A better programming environment would allow for a continuous spectrum ranging from dynamically varying requests with less efficient execution to fixed requests with more efficient execution so that the system can provide the best possible solution for the requirements of the problem to be solved.

*Abstraction*

The concept of data abstraction has received much attention in the programming language community. Recently, there has been interest in the use of data abstraction with respect to data base systems.[18] Many data base systems support abstraction related facilities such as views, integrity constraints and triggers.[3,12,19,33] For example, a view provides a form of abstraction in that the user can construct queries for relations which do not actually exist but are materialized from a number of relations. If the organization of the relations is changed, the view can be maintained by redefining it in terms of this new organization. In addition, some authors have proposed mechanisms for specifying higher level operations on data bases (such as "hire" and "fire").[18] Although some systems offer these abstraction

* Requests can be changed at run-time (e.g., in EQUEL relation and domain names can change dynamically).

facilities, they have not yet integrated them with their counterparts in data abstraction from programming languages.*

## AN INTEGRATED APPROACH

We will now consider the integration of data base objects and operations into a programming language. We will not attempt to give a complete language proposal here. Instead, we shall illustrate some of the benefits of this approach. Our discussion is divided into four areas: type system, data base operations, compilation, and abstraction.

### *Type system*

An extremely important aspect of the language is the extension of the type system to provide access to data base relations residing on secondary storage. While a complete discussion of the type system is beyond the scope of this paper, we will discuss a few aspects of the type system to demonstrate the usefulness of an integrated approach. We have been strongly influenced in our thinking about types by the Mesa language.[15]

In introducing data types for relations in the programming language we will exploit the obvious correspondence between relations and collections of records in programming languages, namely, a relation may be viewed as an unordered collection of records (tuples in data base terminology), where the collection may not contain duplicate tuples. For example,

```
Employee: type=relation of
                    name: string(20),
                    dept: integer,
                    salary: integer,
                    manager: string(20),
                    jobtitle: string(15)
                  end
```

defines Employee as a relation type, where "*relation of*" means "unordered collection with no duplicates of . . .".

A relation type is itself an object upon which operations may be performed to obtain access to types contained within it. Thus,

```
declare e: record(Employee)
```

declares e as a variable whose type is identical to the underlying record type of the relation, and

```
declare deptno: Employee.dept
```

declares deptno to be a variable whose type is identical to the dept column of the relation, i.e. integer.

By allowing the types contained within the relation type to be extracted in this manner, duplication of these types in the program is avoided. Thus, a degree of modularity is achieved. In addition, data integrity is enhanced because the only operations that can be performed on the values in the data base are those allowed on the given type. Because there is only one representation for a value of a given type, whether in the data base or in the program environment, and because the set of types provided in the language is the same as those provided in the data base system, no conversion is needed when data is moved from one environment to the other.

Another important feature of the type system is the ability to specify constant or dynamically varying relation and domain names. For example, the following program fragment declares Rel to be a relation bound to the existing data base relation EMP.

```
begin
  declare Rel: Employee=relation('EMP');
    ⋮
end;
```

If the relation were not constant, e.g., if its name were entered at a terminal, the program fragment would be

```
begin
  declare Rel: Employee,
          Rname: string(15);
  write('What is the relation name?')
  read(Rname);
  Rel: =relation(Rname)
    ⋮
end;
```

where *relation* is a function which takes a string argument and returns a relation as result (or returns *undefined* if no such relation exists). In either case the type of the relation is specified so that operations on Rel can be type-checked, and the program code in the block can be the same. The only change is in the way the relation is bound to Rel. Because the programmer has explicit control over the conversion of the input string into an object of type relation, the existence of the named relation can be tested and an appropriate action taken if it does not exist. This cannot be done easily in any existing language.

The last aspect of the type system to be discussed is type extensibility. Several researchers have mentioned that a type extension facility would be valuable so that types other than the typical primitives (e.g., integers, reals, and character strings) can be placed into a data base record. The proposed language will allow arbitrary fixed length structured objects, except those including pointers or relations, to be a component of a data base record. Providing this feature is not particularly difficult if the data base system supports a typing mechanism which can be extended to include generic functions.*[39]

* Recent work in this direction has been reported [7, 17].

* Generic functions have multiple definitions for an operator where the particular definition invoked at a call is determined by the arguments (e.g., + has several definitions, including definitions for integer, real, complex or mixed arguments).

*Data base operations*

There are several ways that data base operations can be made more convenient, consistent, and more closely related to existing constructs in programming languages. Our interest in this research was motivated by the difficulty one author of this paper had expressing complicated queries in SEQUEL. This led to the design of a procedural query language, described in detail elsewhere.[26] In the following discussion, an overview of the query constructs is presented.

Suppose that we wish to print the names of all employees whose manager is Kathy Fallon. This is written as

```
for x in Rel st x.manager='Kathy Fallon' do write(x.name);
end
```

and read "for each record x in the relation Rel, such that x's manager is 'Kathy Fallon', print x's name."* The variable x is known only in the scope of this statement and is bound to each record in the relation satisfying the qualification. The qualification clause (*st*) is optional.

We believe that this construct is cleaner than those used in many existing data base languages. The binding of language environment variables to data base objects is more consistent and the control structure is more natural, say, than requiring explicit testing of a variable shared by the data base system and the program environment to determine if another record exists.[3]

A query which requires the join of two relations may be written as a nested *for* construct. For example, a list of all employees who earn more than their managers could be obtained by writing

```
for emp in Rel do
  for mgr in Rel st emp.manager=mgr.name do
    if emp.sal>mgr.sal then write(emp.name) fi
  end
end
```

Suppose one wanted to construct a temporary relation having employee records for all people working in department 50. This could be accomplished by using the *for* construct to append qualifying tuples to a (temporary) relation variable.

Alternatively, the *for* statement can be used in its short form

```
begin
  declare Reltemp:employee;
  RelTemp:=[all | Rel:dept=50]
    :
end
```

which says "construct a relation composed of those records in Rel such that dept is 50 and assign it to RelTemp." The construct on the right hand side of the assignment operation is an example of a *relation constructor*. The keyword *all* indicates that all fields of the record are selected. A relation constructor causes a copy of each record to be made. If only the name and salary fields were needed, the expression would be

```
[name,salary | Rel:dept=50]
```

Of course, to assign this value to RelTemp, the type specification for RelTemp must be changed.

The value returned by a relation constructor may contain duplicate records. Such objects are called bags,[27] or multisets.[23] If the value is assigned to a relation variable, then duplicates are removed. However, there are situations in which the value is useful with duplicates retained, e.g., when a function such as average is used as in AVG([salary|rel]). To explicitly remove duplicates from the value, two vertical bars ("‖") are used. A relation constructor may be used directly in a *for* statement without having been previously bound to a relation variable. This is demonstrated by the next example which intermixes query iteration, relation constructors, and functions which take relations as arguments to print a list of all departments and the average salary of employees in that department.

```
for x in [dept ‖ Rel] do
  write(x.dept,AVG([salary|Rel:dept=x.dept]));
end
```

This is read "for x in the set of distinct departments from Rel, print x and the average salary for those employees in department x".

The iterative construct is also used to update individual records as shown in the next example in which all employees are given a 10 percent raise.

```
for x in Rel do
  x.salary:=x.salary * 1.1;
end
```

As in most relational systems, the relation is not actually changed *until all iterations are completed* because the storage structure for the relation may require that records be physically moved when a field is changed, in which case a record could be updated more than once. One solution, used in some set theoretic languages, is to make a temporary copy of the set over which the iteration variable ranges.[13] This would not be practical in this environment because of the size of relations and the time needed to make a copy. Alternatively, the problem may be solved by keeping all changes in a separate file until the end of all iterations.[34] This also solves the problem of keeping a data base consistent after certain kinds of crashes.

*Compilation*

The linguistic benefits provided by an integrated environment will be of little utility in practice unless reasonably efficient code can be produced. Here, we discuss a number of compilation techniques which can be utilized in order to ensure that programs in the language have an efficient realization. The topics discussed are: variable binding time,

* The use of a *for* statement to sequence through a collection of data objects has been proposed before by programming language designers[15,17,21,30,31] but is absent from existing data base languages.

partial evaluation of procedures at compile-time, and program optimization.

### Variable binding time

Binding time is the time at which a variable is bound to an entity, e.g., a specific relation to a variable of type relation. Whether this binding occurs at compile-time or run-time influences both run-time efficiency and program flexibility. Figure 1 shows a graph of binding time of variables in data base queries versus the degree to which these queries are compiled for two relational data base systems.

Compiling is more efficient in execution time and may be more efficient in execution space, while late binding allows more flexibility. Notice that these two systems make explicit commitments, to flexibility (INGRES) or to execution efficiency (SEQUEL). By using compilation and optimization techniques developed for implementing programming languages, it is possible to develop a system that operates in the grey region of the graph. When enough parts of a language construct are fixed (e.g., the relation name is constant) code as efficient as possible can be compiled.[22] Otherwise, less efficient code is compiled. This means that the tradeoff between efficiency and flexibility can be controlled by the applications programmer. Furthermore, it means that only those programs that need more flexibility must pay for it in terms of efficiency. Of course, in those cases where code is compiled to access a specific relation, programs may have to be recompiled if the definition of the relation changes.

### Partial evaluation

A second compilation technique which can be used is the partial evaluation of procedures at compile-time (procedure closure). In SEQUEL, all relation names are viewed as either constants in the program text (in which case compilation is performed) or as strings "read in" at run-time (in which case no compilation is performed). With this organization it is impossible to pass a relation name to a procedure and to compile queries issued from the procedures which use this relation name. However, with global flow analysis techniques[16] combined with procedure closure techniques,[37] a programmer may be given a number of choices as to how such a procedure call should be compiled. These include:

1. compile one procedure body using only the type information about the relation,
2. compile distinct procedure bodies for each of the different (constant) relations passed to the procedure using type information as well as specific information about the relation, or
3. expand certain calls "in-line" so that no call is actually made.

Providing a programmer with a facility such as this gives one considerable control over the tradeoff between execution time and space.

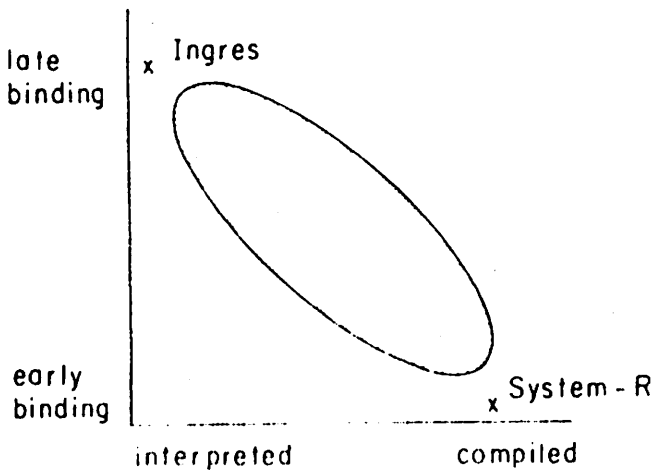


Figure 1 Binding time versus compilation

### Optimization

The use of an integrated language opens up many possibilities for optimization. The goal is to reduce the number of calls on the data base system (and subsequent disk accesses) by combining many queries into one, removing queries from loops, eliminating temporary relations, and so on. This can be achieved through the application of traditional programming language optimization techniques (e.g., code motion, common subexpression elimination, and loop fusion).[1] However, the potential payoff from such optimizations is much greater in data base applications than in typical programs because the removal of a disk access represents a savings many orders of magnitude greater than the removal of, for example, an assignment statement.

Beyond traditional types of optimizations, possibilities exist for more complex optimizations using techniques similar to those used in experimental programming systems which have program verification capabilities.[38] In this case, semantic information available in an abstraction containing relations (e.g., integrity constraints) and information about the physical organization of the data can be used to perform sophisticated optimizations.

### *Abstraction*

Data abstraction facilities have been found useful in developing programs that involve complex data structures.[11,24,40] It seems reasonable to consider extending such facilities to handle relational objects and operations. Another reason for exploring such a facility is because it may provide a structuring mechanism for current data base concepts such as views, integrity constraints, and triggers[3,12,19,33] which are all related to abstraction. Currently these facilities appear in data base systems in an unstructured fashion which often makes it difficult to determine how they interact and relate to one another. A better approach would be to make them all aspects of a single abstraction mechanism.

A view permits a programmer to conceive of a data base in terms of virtual relations. This allows fields, or complete relations, to be operated on even though they might not exist physically. This is similar to the notion of uniform reference that is being explored in programming languages.[14] Another use of views is to prevent a programmer from seeing certain columns of a relation. Again this is similar to a programming language concept (see the *private* attribute in Mesa[15]). A problem with using views is that it is not always clear how a value to be stored to a virtual field should be decomposed and stored into the fields that are physically present. Another problem is that it is not always clear what action should be taken if a value is to be assigned to a field which would cause the record to fall out of the view. Some data base researchers propose to solve these problems by establishing defaults where unambiguous alternatives are possible and disallowing the operations otherwise. An alternative solution, which we are currently investigating, is to acknowledge the difficulty of establishing defaults and instead provide convenient tools for allowing the implementer of the abstraction to specify the semantics explicitly.

Triggers are data base operations which are invoked automatically when certain primitive operations are performed, e.g. when a tuple is deleted from the employee relation the count of employees in the removed employee's department is decremented by one. In the context of a data abstraction facility it would seem more reasonable to associate this operation with the abstract operation "fire" than with the deletion operation itself.

Integrity constraints are assertions which characterize what it means for the data base to be in a correct and consistent state with respect to the semantic meaning of the data. For example, a constraint might be that all the names in the data base must be a subset of those in the employee relation.[19] Currently, such constraint information is used in an enforcement sense, i.e., a constraint is checked after a data base operation (or a group of operations) are performed. Often these checks are very expensive. It may be possible to use this information as assertions to be proved to hold about the set of abstract operations which are permitted on the data base. For those constraints which cannot be proved to always hold it may be possible to automatically determine additional information which should be retained in order to simplify enforcement.

## CONCLUSIONS

We have attempted to show that an integrated approach to providing data base access from a programming language can yield a better programming environment for data base applications than previously available. We are currently in the process of designing a language with this philosophy for use with the INGRES[20] system. Other research groups are also working on similar problems.[5,25,29,36] We believe that this research represents a significant step towards a better understanding of the relationship between programming languages and data base systems.

## REFERENCES


1. Aho, A. V. and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1977.
2. Allman, E., M. R. Stonebraker and G. Held, "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proceedings of a Conference on Data: Abstraction, Definition, and Structure, *SIGPLAN Notices*, 11, Special Issue, March 1976, pp. 25-35.
3. Astrahan, M. M., et al., "System R: Relational Approach to Data Base Management," *ACM Transactions on Database Systems*, 1, 2, June 1976, pp. 97-137.
4. Boyce, R. F., D. D. Chamberlin, W. F. King and M. M. Hammer, "Specifying Queries as Relational Expressions: The SQUARE Data Sub language," *Communications of the ACM*, 18, 11, November 1975, pp. 621-628.
5. Brathergsengen K. and O. Risnes, "ASTRA - A DBMS Based on a High Level, Relational DML with Data Access via a Hierarchical DDL," *Proceedings SIMULA Users Conference*, September 1977.
6. Chamberlin, D. D., et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM Journal of Research and Development*, 20, 6, November 1976, pp. 560-575.
7. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 13, 6, June 1970, pp. 377-387.
8. Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," *Proceedings 1971 ACM-SIGFIDET Workshop*, San Diego, CA, November 1971, pp. 35-68.
9. Codd, E. F. and C. J. Date, "Interactive Support for Non-Programmers," *Proceedings 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Michigan, May 1974.
10. Date, C. J. and E. F. Codd, "The Relational and Network Approach: Comparison of the Application Programming Interfaces," *Proceedings of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Michigan, May 1974.
11. Dahl, O.-J. and C. A. R. Hoare, "Hierarchical Program Structures," in *Structured Programming*, Academic Press, New York, NY, 1972.
12. Eswaren, K. P., "Aspects of a Trigger Subsystem in an Integrated Database System," *Proceedings Second International Conference on Software Engineering*, San Francisco, CA, October 1976, pp. 243-250.
13. Feldman, J., J. Low, D. Swinehart and R. Taylor, "Recent Developments in SAIL—An Algol Based Language for Artificial Intelligence," *Proceedings 1972 Fall Joint Computer Conference*, Vol. 41, Las Vegas, NV, December 1972, pp. 1193-1202.
14. Geschke, C. M. and J. Mitchell, "On the Problem of Uniform References to Data Structures," *IEEE Transactions on Software Engineering*, SE 1, 2, June 1975, pp. 207-219.
15. Geschke, C. M., J. H. Morris, Jr. and E. H. Satterthwaite, "Early Experience with Mesa," *Communications of the ACM*, 20, 8, August 1977, pp. 540-552.
16. Graham, S. L. and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," *Journal of the ACM*, 23, 1, January 1976, pp. 172-202.
17. Gries, D. and N. Gehani, "Some Ideas on Data Types in High-Level Languages," *Communications of the ACM*, 20, 6, June 1977, pp. 414-420.
18. Hammer, M., "Data Abstractions for Data Bases," Proceedings of a Conference on Data: Abstraction, Definition, and Structure, *SIGPLAN Notices*, 11, Special Issue, March 1976, pp. 25-35.
19. Hammer, M. and D. McLeod, "A Framework for Data Base Semantic Integrity," *Proceedings Second International Conference on Software Engineering*, San Francisco, CA, October 1976, pp. 498-504.
20. Held, G., M. R. Stonebraker, and E. Wong, "INGRES—A Relational Data Base System," *Proceedings 1975 National Computer Conference*, 44, Anaheim, CA, May 1975, pp. 409-416.
21. Hoare, C. A. R., "Notes on Data Structuring," in *Structured Programming*, Academic Press, New York, NY, 1972.
22. Katz, R. H., "Compilation of Data Base Queries," M.S. Project, Department of Electrical Engineering and Computer Sciences, U.C. Berkeley, June 1978.
23. Knuth, D. E., *The Art of Computer Programming, Volume 3*, Addison-Wesley, Reading, Mass, 1973.

24. Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, 20, 8, August 1977, pp. 564-576.
25. Merrett, T. H., "Aldat—Augmenting the Relational Algebra for Programmers," Technical Report SOCS-78.1, School of Computer Science, McGill University, November 1977.
26. Prenner, C. J., "A Uniform Notation for Expressing Queries," ERL Memorandum M77/60, Electronics Research Laboratory, U.C. Berkeley, September, 1977.
27. Rulifson, J. F., et al., "QA4: A Language for Writing Problem Solving Programs," *Proceedings IFIP Congress*, 1968.
28. Schmidt, J. W., "Type Concepts for Database Definition: An Investigation Based on Extensions to Pascal," Institut fur Informatik, Universitat Hamburg, June 1977.
29. Schmidt, J. W., "Some High-level Language Constructs for Data of Type Relation," *ACM Transactions on Database Systems*, 2, 3, September 1977, pp. 247-261.
30. Schwartz, J. T., "On Programming: An Interim Report on the SETL Project. Installment 1—Generalities. Installment 2—The SETL Language and Examples of its Use," Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1973.
31. Shaw, M., W. A. Wulf, and R. L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," *Communications of the ACM*, 20, 8, August 1977, pp. 553-563.
32. Smith, J. M. and D. C. P. Smith, "Integrated Specifications for Abstract Systems," Technical Report UUCS-77-112, Computer Science Department, University of Utah, September 1977.
33. Stonebraker, M. R., "Implementation of Integrity Constraints and Views by Query Modification," *Proceedings 1975 ACM-SIGMOD Workshop on the Management of Data*, San Jose, CA, May 1975.
34. Stonebraker, M. R., E. Wong, P. Kreps and G. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, 1, 3, September 1976, pp. 189-222.
35. Stonebraker, M. R. and L. A. Rowe, "Observations on Data Manipulation Languages and their Embedding in General Purpose Programming Languages," *Proceedings Third International Conference on Very Large Data Bases*, Tokyo, Japan, October 1977.
36. Wasserman, A. I. and E. Handa, "Preliminary Report on the Programming Language PLAIN," Medical Information Science, U.C. San Francisco, January 1978.
37. Wegbreit, B., "Procedure Closure in EL1," Center for Research in Computing Technology, Harvard University, March 1972.
38. Wegbreit, B., "Multiple Evaluators in an Extensible Programming System," *Proceedings 1972 Fall Joint Computer Conference*, 41, Anaheim, CA, December 1972, pp. 905-915.
39. Wegbreit, B., "The Treatment of Data Types in EL1," *Communications of the ACM*, 17, 5, May 1974, pp. 251-264.
40. Wulf, W. A., R. L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering*, SE-2, 4, December 1976, pp. 253-265.
41. Zloof, M. M., "Query-by-Example," *Proceedings 1975 National Computer Conference*, 44, Anaheim, CA, May 1975, pp. 431-438.