S O M E   P R I N C I P L E S O F G O O D   L A N G U A G E   D E S I G N

with especial reference to

the design of database languages

C.J.Date

PO Box 2647, Saratoga
California 95070, USA

January 1984

INTRODUCTION

First a position statement. It is my opinion that:

1. Database languages ("query languages") are nothing but special-purpose _programming_ languages.

2. Therefore, the same design principles apply.

3. There are well-established (though not well-documented) principles for the design of programming languages.

4. There is little evidence that current database languages have been designed in accordance with any such principles.

The purpose of this note is to try to pull together a list of such principles, to serve as a reference for database language designers (and any others who might be interested). The list is (obviously) not taken from any one place but rather is culled from a variety of sources, including computer science folklore and "conventional wisdom". One general point that is worth stating at the outset is the following (paraphrased from [2]):

"Most languages are too big and intellectually unmanageable. The problems arise in part because the language is too restrictive; the number of rules needed to define a language increases when a general rule has additional rules attached to constrain its use in certain cases. (Ironically, these additional rules usually make the language _less_ powerful.) ... _Power through simplicity, simplicity through generality, should be the guiding principle._"

ORTHOGONALITY

The "guiding principle" espoused in the foregoing quote is frequently referred to in language design circles as the _principle of orthogonality_. The term "orthogonality" refers to what might better be called "concept independence": _Distinct concepts should always be cleanly separated, never bundled together._ To amplify:

* An example of orthogonality is provided by the PL/I rule that allows any scalar-valued expression to appear wherever a scalar value is required, instead of insisting on (e.g.) a simple identifier in certain specific contexts. The effect of this rule is to allow scalar expressions to be arbitrarily nested; for example, the subscript in a subscripted variable reference can itself consist of a subscripted variable.

* The prize example of _lack_ of orthogonality is provided by the fanset construct of DBTG. (I deliberately use the term "fanset" [1] in preference to the unfortunate term "set" that is used in official DBTG literature.) The fanset construct bundles together at least three concepts and arguably as many as six, viz: (a) a relationship between an owner record and a set of member records; (b) a relationship among a set of member records; (c) certain integrity rules; (d) a set of access paths; (e) a scope for concurrency control purposes; and (f) a scope for authorization purposes. As a result, it may be impossible to drop a fanset that was originally introduced purely for integrity reasons but is no longer needed for that purpose, because some program may now be using it as an access path. This is only one example of the kind of problem that lack of orthogonality can lead to.

The advantage of orthogonality is that it leads to a _coherent_ language. A coherent language is one that possesses a simple, clean, and consistent structure (both syntactic and semantic), a structure that is easy for the user to grasp. If the language is coherent, users are able (perhaps without realizing the fact) to build a simple mental model of its behavior, from which they can make extrapolations and predictions with confidence. There should be no exceptions or unpleasant surprises. In a nutshell, the manuals are thinner, and the training courses are shorter. As reference [5] puts it: "Orthogonal design maximizes expressive power while avoiding deleterious superfluities." (Slightly paraphrased.)

A number of related points arise from the notion of orthogonality:

1. The number of concepts should be small (though not necessarily minimal), in order that the language may be easy to describe, learn, implement, and use. An example of the distinction between "small" and "minimal" is provided by the operations of ordinary arithmetic: Given the operators "infix plus" and "prefix minus", the operators "infix minus",

"times", and "divide by" are allstrictly speaking unnecessary -- but we would not think much of a language that excluded them. As another example, more directly relevant to database languages per se, consider the natural join operator of relational algebra. Natural join is not a primitive operator of the algebra (it is equivalent to a projection of a restriction of a product), and so it may be excluded from a minimal set of relational operators (and is in fact so excluded from many relational languages, including in particular the "structured query language" SQL); however, it is of such overwhelming practical utility that a good case can be made for supporting it directly.

2. A given syntactic construct should have the same meaning everywhere it appears. Counterexample: Consider the two DBTG FIND statements below.

```
    FIND EMP WITHIN DEPT-EMP USING SALARY

    FIND DUPLICATE EMP USING SALARY
```

In the first of these, "USING SALARY" refers to the SALARY field in the User Work Area; in the second, it refers to the SALARY field in the current EMP record. Error potential is high.

3. A given semantic construct should have the same syntax everywhere it appears. Counterexamples: (a) In SQL, the SALARY field of the EMP table is referred to as EMP.SALARY in some contexts, as SALARY FROM EMP in others, and as EMP(SALARY) in still others; (b) in SQL again, rows of tables are designated as "*" in SELECT and COUNT but as blank in INSERT and DELETE (e.g., why is the syntax not DELETE * FROM T ?).

4. Statement atomicity: Statements should be either executed or not executed -- there should be no halfway house. It should not be possible for a statement to fail in the middle and leave the database (or other variables) in an undefined state. Counterexamples: In SQL (at least as implemented in the IBM product SQL/DS), the set-level INSERT, UPDATE, and DELETE statements are not atomic.

5. Symmetry: To quote Polya [3] (writing in a different context): "Try to treat symmetrically what is symmetrical, and do not destroy wantonly any natural symmetry." Counterexamples: In SQL, (a) a GRANT of UPDATE authority can be field-specific, but the corresponding REVOKE cannot, and neither can a GRANT of SELECT authority; (b) in a table that permits duplicate rows, the INSERT and DELETE operations are not inverses of each other.

The following rule is another aspect of symmetry: Default assumptions should always be explicitly specifiable. Counterexample: In SQL, "nulls not allowed" can be explicitly specified (via the NOT NULL clause), but "nulls allowed" (the default) cannot. It is difficult even to talk about a construct if there is no explicit syntax for it.

6. No arbitrary restrictions: While of course it is understood that the implementer may have to impose local restrictions for a variety of pragmatic reasons, such restrictions should not be built into the fabric of the language. Counterexamples: The keyword DISTINCT can appear at most once in a SQL SELECT statement; DL/I supports exactly ten distinct "lock classes" A, B, ..., J; DBTG does not allow a fanset to have the same record-type for both owner and member.

7. No side-effects: No amplification necessary. Major counterexample: Currency indicators in DBTG (which incidentally constitute the absolute linchpin of the DBTG data manipulation language!).

8. Recursively-defined expressions: This does not mean that the language should necessarily support recursion per se -- merely that it should not have artificial restrictions on the nesting of expressions. (This point was touched on earlier.) Counterexample: In SQL, (a) the argument to a function-reference cannot itself be another function-reference, so that (for example) it is impossible in a single statement to compute the sum of a set of averages; (b) a subquery cannot involve a union (and that fact has numerous repercussions, beyond the scope of this short note). See the further discussion of expressions later.

LANGUAGE DEFINITION

The language should possess a rigorous (formal) definition that is independent of any particular implementation. The purposes of that definition are:

1. To provide precise, definitive answers to technical questions and hence to act as the arbiter in any disputes;

2. To (attempt to) prevent the possibility of divergent and incompatible implementations;

3. Generally to serve as _the_ reference source for users, manual writers, teachers, implementers, and anyone else who is concerned with the details of the language at any time.

Counterexamples: Just about every database language in wide usage today. SQL is superior to most in this regard, in that it does (now) possess a formal definition [6], but unfortunately that definition was not produced until after the initial implementation (and that implementation in turn formed the basis of many of the current commercial implementations). Many of the problems with SQL today might well have been avoided if the definition had been produced first.

EXPRESSIONS

In general, the classes of data object supported by a given language fall into a natural hierarchy (or set of hierarchies). For example, relational languages support tables, columns, rows, and scalars, and these object classes can be partially ordered as indicated in the following diagram:

```
             table (highest)
                   |
                   |
             --------------
            |              |
            V              V
         column           row
            |              |
            |              |
             --------------
                   |
                   V
             scalar (lowest)
```

(columns are neither higher nor lower than rows with respect to this hierarchic ordering). Now, completeness dictates that for each class of object it supports, a language should provide at least all of the following:

* a constructor function, i.e., a means for constructing an object of the class from literal (constant) values and/or variables of lower classes (e.g., it should be possible in a relational language to construct a table from a set of arguments represented as row-expressions);

* a means for comparing two objects of the class, at least for equality and possibly also for comparative rank according to some defined ordering;

* a means for assigning the value of one object in the class to another;

* a selector function, i.e., a means for extracting component objects of lower classes from an object of the given class (e.g., it should be possible in a relational language to extract an individual column from a table);

* a general, recursively defined syntax for expressions representing objects of the given class that exploits to the full any closure properties the object class may possess (e.g., if A + B is a valid scalar-expression, then it should be possible to specify the operands A and B as arbitrary scalar-expressions -- it should not be necessary to restrict them to be just simple variable-names).

It is instructive to examine a language such as SQL to see to what extent it meets this "expression completeness" requirement.

MISCELLANEOUS POINTS

I conclude this short note by listing some additional design principles and objectives, taken from a variety of sources (as indicated), without however offering any additional comments.

From the definition of Algol 68 [5]:

* Security

  "Most syntactical and many other errors [should be easily detectable] before they lead to calamitous results. Furthermore, the opportunities for making such errors [should be] greatly reduced."

* Efficiency

  This point includes:

  - Static type checking
  - Type-independent parsing
  - Independent compilation
  - Loop optimization
  - Representation (support for multiple character sets)

From an evaluation of a number of proposals for incorporating database functions into COBOL [4]: [Any COBOL database language should:]

- Be a natural extension of COBOL
- Be easy to learn
- Conform to a standard
- Support relations, hierarchies, and networks
- Promote quality programming
- Be usable as a query language
- Provide set-level access
- Have a stable definition
- Increase programmer productivity
- Be data independent
- Reflect user input in its design

From a proposal of my own for extending the conventional high-level ("host") languages to include database functions [1]:

- [The language] should be designed from the user's point of view rather than the system's (i.e., design should proceed from the outside in).

- It should fit well with the host language. Wherever possible it should exploit existing language features rather than introducing new ones.

- It should transcend and outlive all features of the underlying hardware and software that are specific to those systems.

- It should provide access to as much of the function of the underlying systems as possible without compromising on the first three objectives.

- It should establish a stable long-range design that can be gracefully subset for short-range implementation.

- It should be efficiently implementable.

REFERENCES

1. C.J.Date. "An Introduction to the Unified Database Language (UDL)." Proc. 6th International Conference on Very Large Data Bases (October 1980).

2. R.Morrison. S-Algol Reference Manual. Internal Report CSR-80-81, Dept. of Computer Science, University of Edinburgh (February 1981).

3. G.Polya. How To Solve It. Princeton University Press (2nd edition, 1971).

4. SHARE DBMS Language Task Force. "An Evaluation of Three COBOL Data Base Languages -- UDL, SQL, and CODASYL." Proc. SHARE 53 (August 1979).

5. A. van WijnGaarden et al (eds.). Revised Report on the Algorithmic Language Algol 68. Springer-Verlag (1976).

6. X3H2 (American National Standards Database Committee). Draft Proposed Relational Database Language. Document X3H2-83-152 (August 1983).