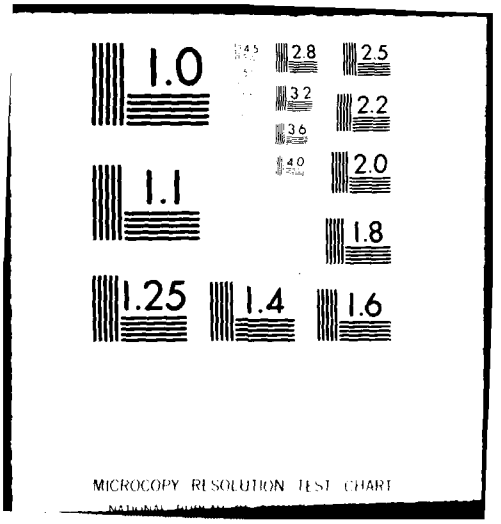
AD-A093 440 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
A CODASYL INTERFACE FOR PASCAL AND ADA.(U)
NOV 80 P BUNEMAN, L MENTEN, D ROOT N00014-75-C-0462
UNCLASSIFIED 80-11-07 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 80-11-07	2. GOVT ACCESSION NO.: AD-A093440	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): A CODASYL INTERFACE FOR PASCAL AND ADA	5. TYPE OF REPORT & PERIOD COVERED: Technical/ 4/80-3/81
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Peter Buneman, Larry Menten, David Root	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Decision Sciences, The Wharton School, University of Pennsylvania, Phila., PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research	12. REPORT DATE: November 1980
	13. NUMBER OF PAGES: 21
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;

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)

codasyl interface, Pascal, ADA, codasyl applications, coding;data currency global data areas, checks at compile-time.

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

A Codasyl interface has been constructed for Pascal and designed for Ada that exploits the data type systems of these languages. We believe that the form of this interface will simplify the writing of Codasyl applications and greatly reduce errors in coding. In particular, it relieves the user from the need to consider data currency and global data areas and uses the host language's type system to perform many checks at compile-time that in other interfaces can at best be done at run-time.

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)

A Codasyl Interface for Pascal and Ada

Peter Buneman, Larry Menten

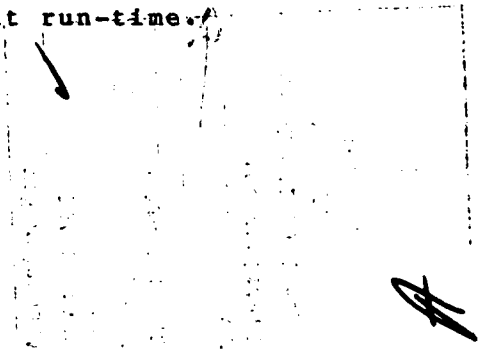
University of Pennsylvania

David Root

International Data Base Systems

Abstract

A Codasyl interface has been constructed for Pascal and designed for Ada that exploits the data type systems of these languages. We believe that the form of this interface will simplify the writing of Codasyl applications and greatly reduce errors in coding. In particular, it relieves the user from the need to consider data currency and global data areas and uses the host language's type system to perform many checks at compile-time that in other interfaces can at best be done at run-time.



Authors' adresses: Larry Menten and Peter Buneman, Department of Computer Science, University of Pennsylvania, Philadelphia, Pa 19104
David Root, International Data Base Systems, 2300 Walnut Street, Philadelphia, Pa 19103

1. Introduction

While the title of this paper may suggest a topic whose practical importance is undeniable but whose technical content is at best mundane, there are a number of advantages that result from building an interface between "strongly typed" languages and a Codasyl system in a fashion that deviates somewhat from that suggested by the DBTG [2] specifications. The interface that is described here, and that has been implemented for Pascal, will, we believe, greatly simplify the process of writing complicated applications programs against Codasyl systems, both by making them more compact and by exploiting those features of Pascal and Ada that are intended to be of assistance in writing correct programs.

The most important feature of this interface is the complete disappearance of the notion of data currency. According to the Codasyl standard, a program communicates with the DBMS through a fixed global area of storage known as the User Working Area (UWA), an area which contains storage for one member of each record type in the database. The DBMS also maintains a global set of currency pointers, physical addresses through which storage and retrieval of records is controlled. The data manipulation routines, which transfer data to and from the database, operate by examining and modifying the UWA and the currency pointers. A programmer who does not completely understand the effects of these routines is likely to find that a record in the UWA has been accidentally overwritten or that an iteration goes wrong because a currency pointer has been reset. Olle [6] gives a good account of the

difficulties involved in manipulating the UWA. For readers unfamiliar with the problems of data currency, they are nothing more than the problems of writing subroutines that communicate with one another through a limited set of global variables; a familiar example may be found in assembly language programming where one must frequently save and restore the registers at the entry and exit of a subroutine.

The other goals we had in building such an interface were to achieve a natural representation of Codasyl structures within the type system of the host language and to exploit, as far as possible, the compile-time type checking that is available. The second point is particularly relevant for the Ada interface, where it is possible to detect during compilation many errors that in conventional Codasyl programming environments do not appear until run-time.

Our decision to investigate Pascal [3] and Ada [1] was motivated for purely practical reasons. It is possible that similar, and perhaps cleaner interfaces could also be designed for other languages with sophisticated type constructs. We are also aware of an effort to design a database extension to Ada based upon DAPLEX [7], and some such extension is clearly needed. However, an interface for existing database management systems is also needed and our current implementation for Pascal amounts to a few hundred lines of code. This interface operates with the SEED [4] database management system, but could readily be rewritten for any other Codasyl system; and only minor changes will be required to create the Ada interface.

2. The Pascal Interface

The Pascal interface consists of two components. The first is a program, an extension to the program that compiles sub-schema definitions, that takes Codasyl data definition language as input and converts it into the appropriate Pascal type declarations, which may then be incorporated in the user's program. The second is a set of database access routines that are declared as external and provide the "data manipulation language" for Pascal. The operation of the database access routines may be described through some simple examples that operate on a database containing information about students, courses and enrollments. Leaving aside, for the moment, the details of the type declarations, the program to print the NAMEs and GRAD_DATEs of all STUDENTs would be:

```
 1.  var S: REC_STUDENT; D: DBREF{STUDENT};
 2.
 3.  begin
 4.    D:=FINDFA(STUDENT);
 5.    while D <> 0 DO
 6.      begin
 7.        GET_STUDENT(D,S);
 8.        WRITELN(S.NAME, S.GRAD_DATE);
 9.        D:=FINDNA(STUDENT,D)
10.      end
11.  end.
```

Figure 1. A simple traversal of a record class

On line 1 of this example, S is declared to be a STUDENT record, and D is declared as a database reference (DBREF), a physical database address. The {STUDENT} comment following the DBREF declaration indicates our intention to use D as a physical address for a STUDENT record. Line 4 sets D to reference the first student in the database (FINDFA stands for find-first-in-area).

Line 8 instantiates S as the record referenced by D; and line 9 (find-next-in-area) generates the DBREF for the next student in the database. Note that in line 9, the DBREF of the previously found student is a parameter for the function FINDNA. It is this simple technique that largely avoids the data currency problem and allows several procedures to traverse the same record class simultaneously and without interference.

It should be noted that, as a result of our natural* desire not to modify the Pascal compiler, the intention of the user that D should be a DBREF for a STUDENT record is not checked at compile-time. An error will however be generated by the database interface at run-time if GETREC attempts to instantiate S with something other than a STUDENT record. In the Ada interface described below, the use of generic routines permits a much greater degree of compile-time checking.

A more interesting use of the Pascal interface is based upon the standard "academic" schema in figure 2.

---

* Although this desire may be justly attributed to the authors' indolence, there would be a serious problem of transportability were such modifications to be made. The present implementation should work against any Pascal system that allows a linkage to external subroutines.

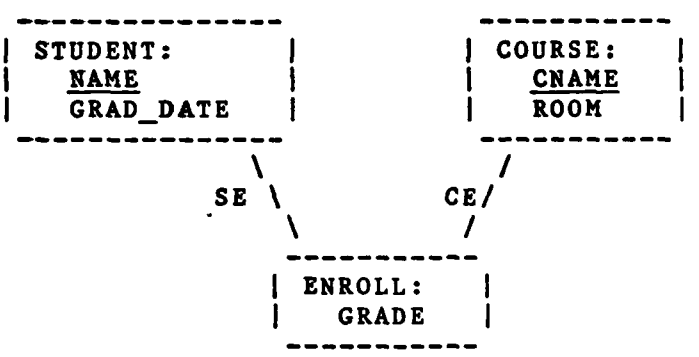


Figure 2. The STUDENT - COURSE schema.

The Pascal code in figure 3 is a (not neccessarily efficient) function that determines whether or not two students are enrolled in the same course. The functions that control Codasyl set traversal are FINDFS (find first in set), FINDNS (find next in set) and FINDOS (find owner).

```
 1.  function SAME_COURSE(S1, S2: DBREF{STUDENT}):BOOLEAN;
 2.  var E1, E2: DBREF{ENROLL};
 3.      C     : DBREF{COURSE};
 4.      FOUND : BOOLEAN;
 5.  begin
 6.    FOUND:=FALSE;
 7.    E1:=FINDFS(SE, S1);
 8.    while (E1 <> 0) AND NOT(FOUND) do
 9.      begin
10.        C:=FINDOS(CE, E1);
11.        E2:=FINDFS(SE, S2);
12.        while (E2 <> 0) AND NOT(FOUND) do
13.          begin
14.            if C = FINDOS(CE, E2) then FOUND:=TRUE;
15.            E2:=FINDNS(SE,E2)
16.          end;
17.        E1:=FINDNS(SE,E1)
18.      end;
19.    SAME_COURSE:=FOUND
20.  end
```

Figure 3. Traversing Codasyl sets.

For example, line 7 of figure 3 establishes the first ENROLL record "owned by" S1. The iteration of lines 8 to 18 repeatedly finds the "owning" COURSE in the set CE (line 10) and then finds

the next of S1's enrollments (line 17). Note that 0 is returned to indicate that a set is exhausted.

Anyone who has written conventional Codasyl routines with explicit manipulation of the UWA will be aware that these examples are considerably shorter and more "structured". Writing the function SAME_COURSE in a conventional Codasyl programming system would, for example, call for the set currencies to be saved after each use of the routines FINDFS and FINDNS and restored for the next corresponding FINDNS. Moreover, all the currencies that could be affected by this function would have to be saved on entry to the function, and restored on exit. Another benefit of this interface is that the not uncommon problem of recursive traversal of Codasyl structures is greatly simplified because the programmer may write recursive Pascal programs rather than build explicit stacks.

In order to represent database update, we should first examine in more detail the type declarations generated for the Pascal interface. Figure 4 shows the data definition language that would be used to define our academic data base, and figure 5 shows the Pascal type and procedure declarations that are generated from it.

```
 1.  Schema name is SCHOOL.
 2.  Record name is STUDENT
 3.        key is using NAME.
 4.     NAME type is character 30.
 5.     GRAD_DATE type is character 6.
 6.  Record name is COURSE
 7.        key is using CNAME.
 8.     CNAME type is character 30.
 9.     ROOM type is character 5.
10.  Record name is ENROLL.
11.     GRADE type is fixed.
12.  Set name is SE
13.     owner is STUDENT
14.     member is ENROLL
15.     set selection is thru locaion mode of owner.
16.  Set name is CE
17.     Owner is COURSE
18.     member is ENROLL
19.     set selection is thru current of set.
```

Figure 4. Part of the Data Definition Language for figure 2.

```
const
   STUDENT     = 'STUDENT                              ';
   COURSE      = 'COURSE                               ';
   ENROLL      = 'ENROLL                               ';
   SE          = 'SE                                   ';
   CE          = 'CE                                   ';

type
   DBREF  = INTEGER;
   DBID = PACKED ARRAY [1..30] OF CHAR;

   TYP_NAME        = PACKED ARRAY [1..30] OF CHAR;
   TYP_GRAD_DATE   = PACKED ARRAY [1..6] OF CHAR;
   TYP_CNAME       = PACKED ARRAY [1..30] OF CHAR;
   TYP_ROOM        = PACKED ARRAY [1..5] OF CHAR;
   TYP_GRADE       = INTEGER;

   REC_STUDENT = RECORD              { Student Records }
                    NAME      : TYP_NAME;
                    GRADDATE : TYP_GRAD_DATE;
                 END;

   REC_COURSE = RECORD               { Course Records }
                   CNAME : TYP_CNAME;
                   ROOM  : TYP_ROOM;
                END;

   REC_ENROLL = RECORD           { Enrollment Records }
                   GRADE : TYP_GRADE;
                END;
   ENV_ENROLL = RECORD     {Environment record for update}
                   NAME: TYP_NAME;
                   CE: DBREF;
                END;

{ EXTERNAL PROCEDURES FOR DBMS INTERFACE }
   function FINDFA(N: DBID): DBREF; EXTERNAL;
   function FINDNA(N: DBID; D: DBREF): DBREF; EXTERNAL;
   function FINDFS(N: DBID; D: DBREF): DBREF; EXTERNAL;
   function FINDNS(N: DBID; D: DBREF): DBREF; EXTERNAL;
   function FINDOS(N: DBID; D: DBREF): DBREF; EXTERNAL;
   function FFC_STUDENT(K: TYP_NAME): DBREF; EXTERNAL;
   function FFC_COURSE(K: TYP_CNAME): DBREF; EXTERNAL;

   procedure GET_STUDENT(D: DBREF; VAR R: REC_STUDENT): EXTERNAL;
   procedure GET_COURSE(D: DBREF; VAR R: RECCOURSE): EXTERNAL;
   procedure GET_ENROLL(D: DBREF; VAR R: REC_ENROLL): EXTERNAL;

   function STO_STUDENT(R:REC_STUDENT): DBREF;
   function STO_COURSE(R:REC_COURSE): DBREF;
   function STO_ENROLL(R:REC_ENROLL; E: ENV_ENROLL): DBREF;
```

Figure 5. Pascal Declarations

The declarations in figure 5 are generated automatically from the data definition language of figure 4. They are to be incorporated into the user's programs by whatever mechanisms (editing, "include" statements etc.) are appropriate. Several of the declarations needed to provide for update and other esoteric features of Codasyl have been omitted from this figure. A few comments are in order:

1. The constant declarations are all for objects of type DBID (database identifier). The database management system for which the interface is implemented is able to use a character string, the name of the record class or set, as such an identifier. In general these identifiers should be whatever is required, often an integer, by the database management system at run-time.

2. The TYP_... declarations are useful for writing further procedures that operate on items in the database. If the item names are not constrained to be unique, some other lexical mapping will be required.

3. The ENV_... record is necessary to establish the appropriate set linkages when storing records. Since STUDENT and COURSE are not members of any sets, they do not require ENV_... records. The ENV_ENROLL record is set up to contain the appropriate information to relate each ENROLL record being stored to a STUDENT record and to a COURSE record. Note that the STUDENT record will be identified by a value for NAME, while the COURSE record

will be identified by a currency value. This currency value could point to either a COURSE record or an ENROLL record (as it then points indirectly to a COURSE record, the one that owns the ENROLL record).

4. The functions FFC_... are for calc key access to records. They return the DBREF for the first record (if any) with the given key. There is also a set of FNC_... functions (not listed) that find other records that have the same key.

5. The STO_... functions are record storing functions. They take an ENV_... record where appropriate and return the DBREF for the stored record. The latter is often useful for subsequent stores.

Figure 6 shows an example of a procedure which stores an ENROLL record. The procedure takes as arguments the name of the student, the name of the course, and the grade.

```
 1.  function STOREENR(STUNAME: TYP_NAME;
 2.                        CRNAME: TYP_CNAME;
 3.                        CRRGRADE:TYP_GRADER): DBREF;
 4.   var E: REC_ENROLL;
 5.          EV: ENV_ENROLL;
 6.          C: REC_COURSE;
 7.   begin
 8.     EV.NAME := STUNAME;
10.     EV.CE := FFC_COURSE(CRNAME);
11.     E.GRADE := CRGRADE;
12.     STOREENR := STO_ENROLL(E,EV)
13.  end;
```

Figure 6. A record storing example.

Line 9 of this figure locates the desired COURSE record using a value for CNAME as a hash key. To pass FFC_COURSE a value for

CNAME we are passing an entire COURSE record. This was done to avoid having to declare a structure for each hash key which would be made up of a dummy item type followed by the hash key item type.

The profusion of procedure names (there must be a STO_... and a GET_... function for each record class) is a problem that can be neatly solved with overloading. It would be possible to have similarly created a different set traversing function for each set. Doing so would permit a greater degree of compile-time checking since the DBREFs could be typed by the record type to which they refer. The problem is that Codasyl exploits the run-time availability of type information. For example, it is possible to ask for the first record (of any type) in a given area; it is also possible to have a set in which the member records are of more than one type. To cope with this we have added

```
procedure DBTYPE(D:  DBREF;  VAR N:  DBID);
```

that sets N to be the DBID for the class of the record to which D refers. Our Pascal representation is therefore a compromise in which we have given the user the full power of the Codasyl run-time system while trying to map Codasyl structures as naturally as possible into Pascal data types.

3. The Ada Interface

The Ada interface is designed upon similar lines to the Pascal interface but there are some important differences. In the first place, access to the database is through a package. This both simplifies the use of the database since it may now be a separately compiled unit and provides the appropriate visibility restrictions on the identifiers used in the construction of the interface. A greater degree of type checking is also provided. The type REF_STUDENT is a database reference that is constrained to refer to a STUDENT record; and the Ada compiler will check, as far as is possible, that the correct types have been used. For example, a compile-time error would be generated on line 4 if the argument and result of GET were not REF_... and REC_... types for the same record class.

```
1.    S: REC_STUDENT;
2.    D: REF_STUDENT:=FINDFA;
3.    while D not NULLREF(D) loop
4.      S:=GET(D);
5.      PUT(S.NAME & S.GRAD_DATE & NEWLINE);
6.      D:=FINDNA(D);
7.    end loop
```

Figure 7. A simple Ada program fragment.

Figure 8 also illustrates this point. The set traversal functions FINDFS, FINDNS and FINDOS are checked at compile-time to make sure that the REF_ types they are given are consistent with the sets being traversed. We believe that this degree of type checking will prove of considerable advantage in writing applications programs against Codasyl systems. In the authors' experience, the run-time availability of type information is

seldom exploited, and the Codasyl programmer usually implicitly types his currency variables. There are of course exceptions to this. Building a general-purpose interactive query language would call for interpreted type information. Another problem lies in Codasyl sets that can contain more than one record type as member. The correct approach here is to construct the appropriate variant record type, discriminated by the name of the record class, an object of type DBID.

```
function SAME_COURSE(S1, S2: REF_STUDENT)
                              returns BOOLEAN is
E1, E2: REF_ENROLL;
C     : REF_COURSE;
begin
  E1:=FINDFS(SE, S1);
  while not NULLREF(E1)  loop
    C:=FINDOS(CE, E1);
    E2:=FINDNS(SE, S2);

     -- etc.

end SAME_COURSE;
```

Figure 8. Traversing Codasyl sets.

For convenience, the database access routines are all overloaded by the types of their arguments and results. For example, there are three internal deifinitions of the function FINDFA, one for each record class that may be returned. Overloading and the mechanism for passing parameters by association may be used to simplify storage and update. In figure 9, for example, the call to STORE is given with parameters named by the appropriate field names.

```
procedure STOREENR(STUNAME : in TYP_NAME;
                   CRNAME  : in TYP_CNAME;
                   CRGRADE : in TYP_GRADE;
                   STUREF  : out REF_STUDENT) is
   ER := REC_ENROLL;
   CREF : REF_COURSE;
begin
  ER.GRADE := CRGRADE;
  STORE(REC := ER; NAME := STUNAME;
        CE := FINDFC(CRNAME); REF := STUREF);
end STOREENR;
```

Figure 9. A record storing example.

The package declaration for this database is quite lengthy and has been relegated to an appendix. Much of this bulkiness results from the definitions needed to overload the database access functions. The amount of nomenclature is considerably reduced in comparison with the Pascal declarations. For the most part the declarations follow those of the Pascal type and procedure definitions given earlier, but a few points should be noted.

1. In the Ada specifications, there is a statement that function subprograms should not do i/o. Strictly speaking, the database access functions will read from the database and should therefore be defined as procedures. However, since data currency has been successfully hidden from the user, there will be no logical side effects resulting from any of the access (as opposed to update) subprograms. We feel that the database should be viewed as a single global structure, and that a database access may correctly be viewed as a function that returns some component of this structure.

2. The creation of the ID_... subtypes is a device to create a set of data types each of which contains only one object. For example the only object of type ID_STUDENT is STUDENT. This allows us to pass in the set name as an argument and to have the appropriate compile time checks made. Only occasionally do two Codasyl record classes have more than one set connecting them. Therefore, a call of the form FINDOS(MEMBER:=...) is usually sufficient to specify which set is involved. In such cases some extra economies might be achieved by further overloading FINDOS as a single argument function.

3. The generic functions defined in the package body are not neccessary; each access subprogram could have been defined by a call to the external database subroutine. However, the use of generic instantiation may improve readability in cases where the database schema is large.

4. Discussion

An attempt has been made to represent Codasyl structures within the type system of two strongly typed languages. While we believe the representation described here has certain advantages, we cannot claim that it is entirely natural. One problem is created by the run-time availability of type information in Codasyl. Another is the profusion of names needed to specify the various types. If Ada had some method for specifying generic types, we could have defined REC(...) and REF(...) as

parameterized types that operate on a DBID such as STUDENT to create the appropriate record and reference types. To some extent this is possible in languages [5,8] in which types can be parameterized. However, one would ideally wish to parameterize not just by the names of the types but also by the relationships among them (such as the relationship between a set and the owner class), and we know of no language that allows this to be done in any generality.

Appendix

Ada package definition

```
package SCHOOLSUB is
   type DBID is private;
   subtype ID_STUDENT is DBID;
   subtype ID_COURSE is DBID;
   subtype ID_ENROLL is DBID;
   subtype ID_CE is DBID;
   subtype ID_SE is DBID;
   STUDENT : constant ID_STUDENT;
   COURSE  : constant ID_COURSE;
   ENROLL  : constant ID_ENROLL;
   CE      : constant ID_CE;
   SE      : constant ID_SE;

   type DBREF is private;
   subtype REF_STUDENT is DBREF;
   subtype REF_COURSE is DBREF;
   subtype REF_ENROLLMENT is DBREF;

   type TYP_NAME is STRING(1..30);
   type TYP_GRAD_DATE is STRING(1..6);
   type TYP_CNAME is STRING(1..30);
   type TYP_ROOM is STRING(1..5);
   type TYP_GRADE is INTEGER;

   type REC_STUDENT is
     record
       NAME: TYP_NAME;
       GRAD_DATE: TYP_GRAD_DATE;
     end record;
   type REC_COURSE is
     record
       CNAME: TYP_CNAME;
       ROOM: TYP_ROOM;
     end record;
   type REC_ENROLL is
     record
       GRADE: TYP_GRADE;
     end record;

   function NULLREF(REF: DBREF) returns BOOLEAN;
   function FINDFA: REF_STUDENT;
   function FINDFA: REF_COURSE;
   function FINDFA: REF_ENROLL;
   function FINDNA(REF: REF_STUDENT) returns REF_STUDENT;
   function FINDNA(REF: REF_COURSE) returns REF_COURSE;
   function FINDNA(REF: REF_ENROLL) returns REF_ENROLL;
   function FINDFS(SET: ID_SE; OWNER: REF_STUDENT) returns REF_ENROLL;
   function FINDFS(SET: ID_CE; OWNER: REF_COURSE) returns REF_ENROLL;
   function FINDNS(SET: ID_SE; PRIOR: REF_ENROLL) returns REF_ENROLL;
   function FINDNS(SET: ID_CE; PRIOR: REF_ENROLL) returns REF_ENROLL;
```

References

1. Ada Preliminary Reference Manual and Design Rationale, SIGPLAN notices, vol 14, 6 (June 1979).

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

3. Jensen, K. and N. Wirth, Pascal User Manual and Report, Springer (1977).

4. Gerritsen, R., Seed Reference Manual, International Data Base Systems, Philadelphia (1978).

5. Liskov, B., A. Snyder and R. Atkinson, "Abstraction Mechanisms in CLU" Comm. ACM, vol 20, 8, pp.654-576 (August 1977).

6. Olle, P.W. The Codasyl Approach to Database Management, Wiley (1978).

7. Shipman, D. "The Functional Data Model and the Data Language DAPLEX". ACM Transactions on Database Systems (to appear).

8. Wulf, W.A., R.L. London, and M. Shaw, "Abstraction and Verification on Alphard...", Technical Report, USC, Los Angeles (1976).

```
   function FINDOS(SET: ID_SE; MEMBER: REF_ENROLL) returns REF_STUDENT;
   function FINDOS(SET: ID_CE; MEMBER: REF_ENROLL) returns REF_COURSE;

   function FINDFC(NAME: TYP_NAME) returns REF_STUDENT;
   function FINDFC(CNAME: TYP_CNAME) returns REF_COURSE;

   function GET(REF: REF_STUDENT) returns REC_STUDENT;
   function GET(REF: REF_COURSE) returns REC_COURSE;
   function GET(REF: REF_ENROLL) returns REC_ENROLL;

   procedure STORE(REC: in REC_STUDENT;
                   REF: out REF_STUDENT);
   procedure STORE(REC: in REC_COURSE;
                   REF: out REF_COURSE);
   procedure  STORE(REC: REC_ENROLL;
                    NAME: TYP_NAME; CE: REF_COURSE;
                    REF: out REF_ENROLL);

private
   type DBID is STRING(1..30);
   STUDENT : 'STUDENT                    ';
   COURSE  : 'COURSE                     ';
   ENROLL  : 'ENROLL                     ';
   SE      : 'SE                         ';
   CE      : 'CE                         ';
   type DBREF is
     record
       KIND: constant (NIL, NOTNIL);
       when NIL => null;
       when NOTNIL =>
         constant DBTYPE : DBID;
         constant VALUE  : INTEGER;
     end record;

end SCHOOLSUB
```

```
package body SCHOOLSUB is

  function XFINDFA(ID: DBID) return DBREF is
  begin
    pragma INTERFACE(CODASYL);
  end XFINDFA;

  generic (type T; ID: DBID)
  function GFINDFA return T is
  begin
    return XFINDFA(ID);
  end GFINDFA;

  function FINDFA is new GFINDFA(REF_STUDENT, STUDENT);
  function FINDFA is new GFINDFA(REF_COURSE, COURSE);
  function FINDFA is new GFINDFA(REF_ENROLL, ENROLL);

  -- elaboration of FINDNA

  function XFINDFS(ID: DBID; REF: DBREF) return DBREF is
  begin
    pragma INTERFACE(CODASYL);
  end;

  generic (type IDTYPE; type OWNERTYPE;
              type MEMBERTYPE; ID: DBID)
  function GFINDFS(SET: IDTYPE; MEM: MEMBERTYPE)
                        return OWNERTYPE is
  begin
    return XFINDFS(ID, MEM);
  end GFINDFS;

  function FINDFS is new GFINDFS(ID_SE, REF_STUDENT, REF_ENROLL, SE);
  function FINDFS is new GFINDFS(ID_CE, REF_COURSE, REF_ENROLL, CE);

    -- etc

end SCHOOLSUB;
```