University of Edinburgh
Department of Computer Science

James Clerk Maxwell Building
The King's Buildings, Edinburgh

University of St Andrews
Department of Computational Science

North Haugh
St Andrews, Fife

Progress with Persistent Programming

Alan Dearle.

PPR-8-84
February, 1984

Progress with Persistent Programming

M.P. Atkinson,
University of Pennsylvania, Dept. of Computer and Info. Science
School of Engineering and Applied Science, Philadelphia, PA 19104

P. Bailey, University of St Andrews, Dept. of Computational Science
North Haugh, St Andrews KY16 8SX, Scotland

W.P. Cockshott,
University of Edinburgh, Dept. of Computer Science,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland

K.J. Chisholm,
University of Edinburgh, Dept. of Computer Science

R. Morrison,
University of St Andrews, Dept. of Computational Science

DATA CURATOR
DOCUMENTATION

10 February 1984

Table of Contents

11. Acknowledgements

List of Figures

The department of Computing and Information Science at the University of Pennsylvania has provided the support and stimulus for the preparation of this document. We would particularly like to thank the Data Base Programming Language Seminar Class of the Fall of 1983, who collected much of the data and behaved as a sounding board for all of it.

It is inevitable that we will have misunderstood some people's languages, partly due to insufficient care, but often because the papers we could obtain were obsolete or gave insufficient detail. In these cases we apologise to the reader for our inaccuracy and to the original authors for our misrepresentation. Similarly the selection of one language for detailed study rather than another may sometimes be arbitrary - in which case we apologise to those authors whose work was omitted - and say to them, please be sure we have your latest papers for the revised version. To the reader the remedy is to wait for the new edition.

1. INTRODUCTION

In a preceding chapter Peter Buneman has analysed the problems of matching programming languages and databases. This chapter re-examines that domain from another viewpoint, as progress in the relationship between programming languages and databases is crucial to the exploitation of both these software system components.

The various developments which have influenced the way people can program with persistent data are examined from the viewpoint of programming language design. For engineering reasons we cannot ignore feasibility (an understanding of which has been developed by the database community), but in this chapter the emphasis is on linguistic issues.

The approach of language design is used to separate issues so that their independent

development may be understood and assessed, particularly of interest is the issue of how to provide persistence. The development considered includes the early packages of subroutines, the hosted DML's such as EQUEL [Stonebraker *et al.* 76] and those used for CODASYL databases [Olle 78]. These progress to the attempts to build integrated languages by combining database constructs with existing programming languages to form new languages, as in: PASCAL-R [Schmidt 78], PLAIN [Wasserman *et al.* 81] and RIGEL [Rowe 80]. Those languages combine the features of PASCAL and relations; a similar philosophy underlies ADAPLEX [CCA83], where Shipman's version of the functional data model, DAPLEX [Shipman 81], is being embedded in ADA [Ichbiah *et al.* 79]. The view that the principle of data type completeness is of paramount importance has led to the PS-algol experiment which takes a contrasting approach: that the critical step is not the addition of new data structures to a language but the provision of persistence as an orthogonal property of all existing structures.

It is concluded that the treatment of persistence as an orthogonal property of data structures is a promising strategy in the design of database languages. This prompts a number of questions:

- What are the data structures we would like in these languages?
- How should current work in programming languages influence database programming languages?
- How will the need for persistence influence language design?
- As better database programming languages become available, how will programmer behaviour change?
- Will the form of constructed systems change?
- Do these changes require a revision of the programming support environments?

This chapter is an attempt to further identify these questions about language design and to provide our view of the currently available answers.

Our discussion is restricted to the programmer's interfaces to persistent data. One would hope that the general principles of language design will soon influence query languages and other user interfaces. The programmer interface, however, is important and will remain so, as there is always a need to write complex algorithms to operate against some persistent data.

1.1. Structure of this chapter

In section 2 a review of language design is given, as it pertains to integration with databases. Section 3 introduces an illustrative example.

The next two sections are concerned with the history which leads to the present languages, as it has had a marked influence on their form. Section 4 examines the use of collections of procedures to provide database facilities in a language. It is shown to be inadequate, though it has the advantage of flexibility; unfortunately some so-called languages for database applications still depend on this approach. The following section looks at languages made by embedding a DML in an existing language. Again problems fundamental to the approach are exposed.

The next group of three sections examine present research in database programming languages. Integrated programming languages, persistent programming languages and conceptual programming languages have each been allocated a section. The integrated languages are the attempts to take existing data models and existing languages and to derive one coherent whole. The persistent programming research takes the view that data bases can be constructed from the data types in a programming language, and that the task is to regulate the promotion of data to longer term persistence. The conceptual language research is the attempt to find languages that correspond with some *natural* way of organising and manipulating the data. There is overlap between categories; for example, Galileo, presented as a conceptual programming language, is also a persistent language.

Section 9 gives the reader a glimpse of current work in programming language design so that an appreciation may develop of the potential influences. We particularly note succinct notations, simple but precise semantics, and flexible type systems. We also observe a preference for *applicative* or *functional* programming style.

The paper concludes with a synopsis of the issues that have been exposed.

2. LANGUAGE DESIGN

The aims of the programming language designer must serve the cause of making it easier for programmers to write and maintain programs written in the language. Sought after properties are:

- The programs should be easy to read and understand,
- The language should be easy to learn, remember and understand,

- The language should be succinct.

To achieve these goals the language designer has found it beneficial to adopt the following principles [Strachey 67]:

- The principle of data type completeness
- The principle of abstraction
- The principle of correspondence.

The principle of data type completeness states that all data types must have the same "civil rights" and that the rules for using the data types must be complete, with no gaps. This does not mean that all operators in the language need be defined on all data types but rather that general rules have no exception. This principle is important in making the language powerful and in making it easy to learn, understand and remember. The avoidance of special cases leads to simpler yet more powerful languages. For further discussion see [Berry 81, Hoare 73, Richard & Ledgard 77, Tennent 77, Wirth 74].

Abstraction is the process of extracting the general structure to allow the inessential details to be ignored. This principle is invoked by identifying the semantically meaningful syntactic categories and providing abstractions over them. For example:

Syntactic category	Abstraction
expression	function
statement	procedure
declaration	module

The principle of correspondence states that the rules for introducing and using names should be the same everywhere in a program. Often this refers to the rules for declaring names in program blocks, and those for naming the parameters of procedures. But it may equally be applied to the rules for introducing the names of fields in records, or any other names, such as the names of modules.

The importance of locality is partly a responsibility of the language designer (for example in S-algol [Cole & Morrison 82] declarations can immediately preceed the need for the identifier, in contrast to Pascal [Jensen & Wirth 75]), and partly an injunction to use a language wisely. It means that programmers' design decisions should be localised. This was called information hiding by Parnas [Parnas 72]. In practice it means that the

language designer should provide scopes and modules[1], and that the programmer should make his declarations in as local a scope as possible, and group logically closely related code together into modules.

To organise a language design, it appears to be best to decide first what objects the language will operate on. This can be further divided into identifying the atomic objects, and then the constructors which allow composite objects to be synthesised from them. Next, one identifies the operations which may be applied to the objects, identifies the supported abstractions, introduces a store to hold the objects and finally packages these concepts in a simple syntax.

Of course design is never so simple, and the process is not simply ordered, but involves iteration and insight. One important iteration is to improve the parsimony of concepts. If two concepts in the language look similar the designer must investigate whether there is a more primitive concept that will serve both roles.

2.1. Applying the language design principles to database languages

These stages of a design will be used to look further at language features.

2.1.1. The idea of persistence

We use the word *persistence* to describe that property of data that determines how long it should be kept. It is an orthogonal property of data, in that, in principle, any data item may exist for an arbitrarily long time. For our purposes, we equate existence with being potentially accessible by some programmer, or program operation. Thus the idea is already familiar. The data associated with different procedure activations already has variable persistence; that in the innermost activations being most transient, and that in the global scope of the program being most persistent. The data in files and databases has been stored there in order that it may have longer persistence. The property of persistence can therefore be thought of as a continuous variable describing one aspect of that data.

2.1.2. The data objects and the principle of completeness

We first need to identify the objects to be manipulated. Candidate objects have been: sets [Childs 68], rings [Gray 67], networks [CODASYL 71]) and relations [Codd 70]. The newer data models [Shipman 81, Hammer & McLeod 81, Codd 79] also imply candidate objects. But the principle of data type completeness reminds us that the rules governing

[1] most query languages and data description languages fail to do this altogether

their manipulation should be consistent. The following examination of existing languages will show that this principle has not been applied universally. Some data types have been allowed to have only persistent instances, others (those that already existed in a "parent" language) have been allowed only transient instances. We intend to show that adherence to the principle of data type completeness would lead to better database programming languages.

2.1.3. The cost of not providing data type completeness for persistent data

The failure to adhere to this principle has a high cost. Large parts of programs are concerned with making the transition between the two data worlds (persistent and non-persistent). The programmer has to organise a translation, which takes code and CPU time. It has an even higher cost: a typical programming task requires that the programmer understand something in the real world, and construct a model of it in the program. As an example, consider the task of building a set of programs to assist in the design of the combustion chamber in an internal combustion engine. As this involves chemistry, thermodynamics, materials science, turbulent fluid dynamics, and the modelling of complex shapes, it is not a trivial task. Yet it is typical of many programming tasks. Understanding the problem and modelling it in a program is hard enough. Add the requirement that the data stored between design sessions has to be in a different form (a CODASYL database perhaps), and the programmer's intellectual difficulties are much increased. He has to relate the real system not only with his program model, but also with the stored model. He also has to implement and manipulate the mapping from stored model to program model. Thus instead of having to visualise only one mapping, he has to manage three, making his task approximately three times as difficult.

Apart from this incurring large learning and program design costs, it also has very severe maintenance costs. It is easy for different programmers to visualise different relationships between the two data models. Consequently someone undertaking maintenance may easily misunderstand and introduce deep errors into the software - errors which may be manifested much later in corrupt persistent data.

To some extent the present interest in integrity constraints [Date 81] and their centralisation [Nijssen 80] is an attempt to treat the symptoms of this rather than the cause. That approach will never cover all conceptual errors, since it is not feasible to recognise all potential errors, and record rules which prevent operations which may cause such errors without preventing legitimate operations. Even were that *intellectually* feasible, enforcing such a complete set of rules would not be a feasible *engineering* task.

A yet more fundamental objection to not adhering to the principle of data type completeness can be identified. When someone builds a database it can be viewed as a model of the real world [Kent 78]. Similarly when a useful program runs it can be thought of as manipulating a real world model. For example, if a stress analysis program is applied to an airframe, then the data structures modelling the airframe and the loadings on that frame constitute a real world model. What is likely to be useful for one modelling activity is likely to be useful for the other. We would expect that proven modelling techniques in the programming languages (here called data type declarations and abstract data types) would be useful in the database. Similarly, improved concepts in modelling developed for databases would be useful in programming languages. There is, therefore, a sound philosophical basis and good engineering reasons for trying to conflate, or at least reconcile, the development of these two areas.

It is clearly wasteful if different code has to be written and maintained to achieve the same effects - in one case on persistent data, and, in the other, on transient data. We believe that the language should be so defined that a procedure in it may be written without knowing whether it will be supplied with persistent or transient data as the actual values of its parameters. We call this *persistence independent programming* and believe all satisfactory languages will need to support persistence independence.

2.1.4. Summarising data type completeness

The evolution of database languages can be evaluated against this criteria of data type completeness. We note a search for appropriate data types, and see a progression towards consistency in the treatment of these data types particularly with respect to persistence. We expect cross-fertilisation of ideas on appropriate data types in both directions, from language to data model, and from data model to programming language.

2.1.5. Introducing the conceptual store

Many languages make explicit the concept of a store in which to hold representations of the manipulated objects. For example, Fortran introduces store in common blocks and as variables. The Algols have a store associated with the nested activations of scopes. PASCAL [Jensen & Wirth 75] and Algol68 [van Wijngaarden *et al.* 69] have added another store, the HEAP which in one case is explicitly relinquished and in the other is conventionally recovered by garbage collection (Actually this is not strictly a property of the language definition - see Tanenbaum [Tanenbaum 78] for an explanation).

It is not essential that the language should have a notion of a store (and by implication of representations). The applicative languages [Henderson 80, Darlington *et al.* 82] avoid

the whole concept of a store. Notionally they have only values, the results of expressions. This approach seems inappropriate for database applications. It does not seem helpful to visualise recording the change to one person's salary as the creation of a copy of the whole payroll, making a new record for that person in passing. (It is interesting to note, however, that that was the model of persistent storage when data processing depended on magnetic tapes.) The practical implementation of an applicative language must detect and avoid the copying of massive structures; this is not simple. Storage may be used like a blackboard; so that other people can see what you have written when you've written it (consider a reservation system). But when you "write on the blackboard" in an applicative language, you get your own new private blackboard. Thus we reject the notion of not having a store. Both interpretations of the phrase *data sharing* are required: many users should be able to access and see changes to a given item of data, updating it within constraints, and many data items should be able to *refer* to a single instance of a data item, so that updates to it are reflected in all those contexts.

What store should we have? Associated with the store are the properties of addressing structures, accessibility and longevity of data, and concurrency of access. The abstraction of addressing structure in most languages is the identification of data by variable names. Such names may lead to complex structures (arrays, collections of records interconnected by pointers, etc). In the Algols, the scope in which those names occur dictates the longevity of the data: those in outer scopes lasting longer than those in inner scopes. The introduction of the heap, and of the reference or pointer, allows the longevity of the data to differ from the longevity of the scope in which it was created. If we are already alert to the search for a consistent abstraction which includes the database's store, we note that here is an existing range of persistence. The introduction of the database extends that range, so that data may persist for longer than the execution of one program. One can envisage this orthogonal property of data as a continuum from the most transient values constructed within individual instruction executions in the CPU, to data held indefinitely as computer systems come and go. So far, languages have provided data that has one range of persistence, and databases have aimed at the other limit of the continuum.

An alternative view of this provision of persistence is to say that what is sought is an adequate abstraction for the composite store of main memory and backing store devices (predominantly discs). One existing abstraction is that of virtual memory. This abstracts over the properties of size and speed, but not that of longevity. The introduction of mapped files [Organick 72] extended this abstraction into that of

persistence, by linking it with a persistent naming scheme (the filing system). This extension is limited to a timescale where the program does not change to the extent of changing its datastructure definitions - a well known problem in the database range of persistence.

There is a need for a safe and easily understood concurrent use of store in a database programming language. Influences worth considering in this search come from theory: CCS [Milner 79, Milne & Milner 79] and CSP [Hoare 78] and from actual languages: ADA [Ichbiah *et al.* 79], concurrent Pascal [Brinch-Hansen 75], OCCAM [Taylor & Wilson 82] and ARGUS [Liskov, B. *et al.* 83]. The first incorporation of these ideas into linguistic concepts for database programming languages may be ADAPLEX [Computer Corporation of America 83]and the work reported on the plans for a successor to PASCAL/R based on modula-2 [Mall *et al.* 83].

Databases have other notions of store - that different people looking at its contents will see them differently, and will have different entitlements to operate on different parts of the store. Most programming languages have abrogated responsibility for concurrency and protection, leaving it to the operating system which controls data with less precision. But programming languages are approaching that issue now, in the integration of the program development environment and its tools for the assembly of modules into systems, as well as in the module concept itself [Jones & Liskov 78, Lampson 83]. Later examples show how functional abstraction can limit the operations available and provide modified views of data. Again the plans devised by Joachim Schmidt's group [Mall *et al.* 83] incorporate these needs.

2.1.6. Identifying the operations on the objects

It is not helpful to talk of the operations in abstract terms, since they are different depending on the objects they manipulate. The programming languages are predominantly restricted to operations only on their basic data types (integer, reals, strings, etc.) and tend to rely on procedural abstraction to provide operations on the composite data items. In databases, bulk operations are considered useful. For example: select, project, join, group-by for relations [Codd 70, Gray 82, Gray 81], the FOR statement of DAPLEX [Shipman 81], the cascade delete in CODASYL [CODASYL 71], the union, intersection and difference operations on sets etc. Well presented bulk operations would be of benefit to most programmers even when working on transient data.

2.1.7. Identifying useful abstractions

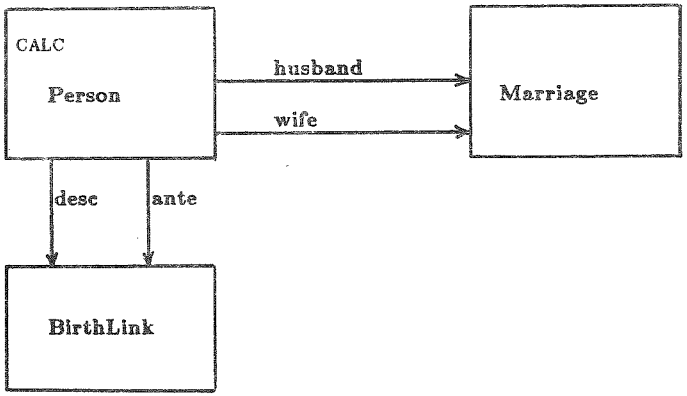
Abstractions hide detail. The three already identified really do not cover the scope of database work. Is the transaction a good abstraction over operations? Do we have an abstraction to cover the ways we identify, access and share data? As yet the required abstractions for using persistent shared data have not become clear. Research is required to identify and refine them.

3. AN ILLUSTRATIVE EXAMPLE

An example has to contain all the factors we wish to illustrate, but has to be small enough to fit in the paper and to be understood. In this case we wish to rework it from a number of approaches, and to make it appear a plausible candidate for using a programming language. Inevitably, therefore, it is contrived. To illustrate aspects of persistent programming it has to be more complex than the rather simple examples often used to illustrate programming languages. It is therefore necessary to explain the test case.

Let us assume that there is a collection of data about people. For each person there is a NAME, the SEX, date of birth (DOB), date of death (DOD), FATHER, MOTHER, SPOUSES and CHILDREN. For example it could be represented by a structure corresponding to the Bachmann diagram in figure 3-1. No assumptions are built in about social mores, but the biological constraint that people have one father and one mother is honoured.

Figure 3-1: A Bachmann diagram of the example data



A possible representation of this data using relations is shown as figure 3-2. Underlining

Figure 3-2: A possible representation of the example data as relations

```
Person (P#:pn,name:s;sex:mf;dob:ymd;dod:ymd)

Parent/Child (Parent:pn;child:pn)

Spouse  (male:pn;female:pn,from:ymd;to:ymd)
```

indicates the key of each relation. A possible representation using the DAPLEX style of functional data model is shown in figure 3-3.

Figure 3-3: A possible representation of the example data in DAPLEX

```
DECLARE Person ( )                          →>   ENTITY
DECLARE name(Person)                        →    STRING
DECLARE sex(Person)                         →    STRING
DECLARE dob(Person)                         →    DATE
DECLARE dod(Person)                         →    DATE
DECLARE father(Person)                      →    Person
DECLARE mother(Person)                      →    Person
DECLARE child(Person)                       →>   Person
DECLARE spouse(Person)                      →>   Person
DECLARE from(Person,Spouse(Person))         →    DATE     !Date marriage
DECLARE to(Person,Spouse(Person))           →    DATE     !starts & ends
```

Given this data examples which illustrate that there will be a continuing need for programming interfaces are:

- *investigate the relationship between generation of antecedent and date of birth, where generation is counted from the most recent decendent known.*
- *investigate the variation in relationship between date-of-marriage of the parents and the date-of-birth of the child.*

Neither of these can be done directly in any user DML we know, though substantial parts can be done in DAPLEX and in CODD [King 82]. Comparable actual computations are well known to the authors in animal breeding research, horse racing adminstration, bill of materials processing, parts explosions, and critical path analysis. We can therefore consider them realistic, but they are over-complex for pedagogic purposes. In fact they are rather simple compared with many engineering and scientific uses of persistent data. The example used here is the determination of all the known antecedents of a given individual. In fact, this can be done directly in DAPLEX as shown in figure 3-4 which is assumed to operate on the data described in figure 3-3. We will treat that query as needing a programming language.

Figure 3-4: A query in DAPLEX to print the antecedents of Janner Curnow

```
DEFINE parent(person)      -> UNION(father(person),mother(person))
DEFINE antecedent(person)  -> TRANSITIVE of parent(person)

FOR THE person SUCH THAT name (person) = "Janner Curnow" DO
  FOR EACH p IN antecedents(person) DO {print (name(p)); print (dob(p))}
```

We are now in a position to define our standard test of database programming languages. It consists of four parts communicating via a body of permanently stored data. These are shown in figure 3-5.

Figure 3-5: A standard test for evaluating Database programming languages

1. *Describe the data necessary to represent the family trees of a set of people.*

2. *Demonstrate simple retrieval by printing the names and dates of birth of all the females. Besides indicating whether simple things can be done simply, this acts as an introduction to many languages.*

3. *Demonstrate* COMPUTED *retrieval by enumerating the names and dates of birth of the ancestors of Janner Curnow, as defined above (figure 3-4).*

4. *Demonstrate update by recording that Janner Curnow and Moyana Trefusis have a daughter, to be called Morwenna Curnow, born 22nd September 1982.*

The final three tests are to be separate programs which operate independently. Ideally the first two should be constrained not to change the data and the final one constrained to biologically reasonable births.

The simple retrieval (test case 2 of figure 3-5) might be coded in DAPLEX as shown in figure 3-6.

Figure 3-6: Retrieval of all the females, coded in DAPLEX

```
FOR EACH p IN person SUCH THAT sex(p) = "F"
  DO {print (name(p)); print (dob(p))}
```


4. CALL INTERFACES TO PERSISTENT DATA

All high level languages have the ability to use a library of separately defined procedures. The earliest attempts to provide a systematic mechanism for persistent data simply involved providing such a library.

The intent was to:

- Make it easier to write programs (the programmer did not need to know about the devices),
- Make it easier to move programs between machines (variations of the environment were hidden within the library).

An evolution in these packages can be seen, and we identify a number of stages:

1. Backing store I/O: These provide the lowest level operations - opening and closing files or tapes, reading or writing blocks.

2. Data structure transfer operations - An example was a system to read/write all (or slices of) Fortran and Algol60 arrays.

3. Software to provide operations on abstract models. An obvious example is software to read/write records properly integrated into the language in the case of COBOL but supplied by libraries in most other languages.

4.1. Backing store operations

Why is there such a progression? The backing store I/O still left the programmer with a great deal he must understand. It also tended to be device and machine specific. It required considerable programming and system skills to arrange to save data structures on disk. The data objects provided and manipulated were possibly an adequate model of what could be done on magnetic tape, but were an inadequate model of the capabilities of disk storage. There, references to data on other parts of the the disk could be stored and exploited to advantage. But with this model, the programmer invented his own scheme for representing and evaluating references. This was often idiosyncratic, making software design, extension, and maintenance difficult, and leading to poor reliability. The access and addressing capabilities of disks lead to more data sharing, so this obscurity of use and lack of reliability became economically serious.

4.2. Data structure libraries

The data structure transfers were the first step in avoiding these problems. By manipulating specified structures, the details of storage were hidden, and the programmer ceased to be responsible for the representation and use of pointers. Taking

our design approach, we must consider what data object types were proposed. They were:

- arrays from Fortran and Algol - chosen because the implementors could then avoid the problems of handing pointers and because engineering applications processed data in arrays.
- associative structures - index sequential files, inverted files and trees chosen because of the utility of associative access in data processing.
- serial structures - files, lists and multithreaded lists of records.
- and ring structures - chosen because they were useful in modelling hierarchies and many-to-many relationships.

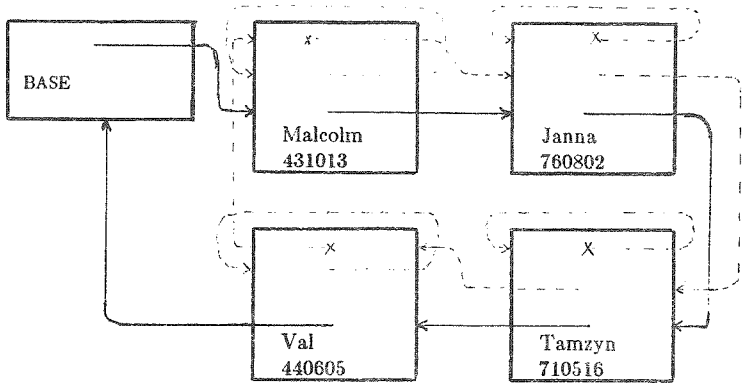
A number of surveys, though not explicitly expressing this viewpoint, report the state of this search for the correct data types in the late 60s [Gray 67, Dodd 69, Atkinson 74]. Programming with these objects was still not satisfactory, but it was a significant improvement over programming with the I/O routines. The I/O routines were not the subject of an example, as it would be obscured by volume and detail. An example of the antecedants retrieval (task 3 of figure 3-5) against a ring structures collection of people is given in figure 4-1. This is an arbitrary choice which mirrors the style of programming used in RSP [Wiseman & Hiles 68] or ASP [Lang & Gray 68], as it might be called from a language like BCPL [Richards & Whitby-Strevans 79].

Figure 4-1: Antecedants Retrieval using a Ring Structure Processor

```
let exampleRetrieval(Name) be $(
   let db = OpenDBfile("Example")        ||mapping from OS names to program
   let x  = GetBaseNode(db)              ||get name from which all else hangs
   let y  = Ring (x, 1)                  ||ring with people on search for person
   while dataPart(y, 1) ≠ Name do        ||is this he?
    $( y := NextOnRing(y, 1)             ||all people threaded on ring 1
      if y = x do                        ||back where we started?
         $(  report("person not found"); return $) $)
||here y refers to the node which represents the person whose
||name is Name and hence whose antecedants are wanted
      writes ("The Antecedants of  "); writes(Name); writeDate (dataPart(y,2));
      writes("are*n"); PrntAnte (y) $)
and PrntAnte(person) be $(               ||a recursive depth first scan of the tree
   let p = Ring(person, 2)               ||parent ring threads as ring 2 in a child
   while p ≠ person do                   ||are we back at the original child?
 $( test RingMark(p) then                ||a parent is marked, a sibling isn't
    $( write(dataPart(p,1)); WriteDate(dataPart(p,2)); newline(); PrntAnte(p)
       p := NextOnRing(p, 3) $)          ||parent ring threads as ring 3 in parents
    or p := NextOnRing(p, 2) $) $)
```


In that example the reader has been spared much practical detail. The problem inherent in this approach is the difficulty of visualising the data representation, yet the necessity of doing so. For example, one needs to know that the ring of all people is the first ring in every node, the complex rules identifying how the parent/child ring threads nodes differently when it passes through them in different roles, the way a parent on a ring is identified and the order in which the data is stored. The imaginary programmer appears to have had this problem in mind when formulating his comments. The representation he has imagined is illustrated in figure 4-2.

Figure 4-2: An instance of the structure being processed by figure 4-1



The intrinsic cause of the problem is that the object types manipulated are not embodied in the language. They are captured somewhere implicitly in the collection of routines; *Ring, dataPart, NextOnRing,* etc. The generic form of instances representing a particular class of objects is nowhere described. This fundamental failure is manifest in there being no way of naming the objects and their parts. (Using identifiers for the constants only gives the impression of naming - the actual association of name and value is at the mercy of the programmer.)

Consider the update operation which is shown in figure 4-3. Again much practical detail has been omitted, such as checking the parents are alive, of opposite sex, and of sensible age at the time. The problems of this implicit model are probably easily visible - even in this very simple case. Note what happens if you insert a person on the wrong ring for their role. This can be done just by mistyping a 2 for a 3. What will the retrieval program do then? The fault would become visible only when someone tries to

Figure 4-3: Adding a new child example using a Ring Structure

```
let exampleUpdate() be $(
let db  = OpenDBfile("Example")
let x   = GetBaseNode(db); StartTransactionOn(x)
let y   = Ring(x, 1)
let JC  = FindPerson(y, "Janner Curnow")
let MT  = FindPerson(y, "Morwenna Trefusis")
let MC  = CreateNode("person"); addDataElement (MC, "Morwenna Curnow")
addDataElement (MC, 19820922); addDataElement (MC, true)

for i = 1 to 4 do addRing(MC)            || set up 4 empty rings
insertOnRing (MC, 1, y)
insertOnRing (MC,2,Ring(JC, 3), true)    ||record father
unless SameRing (Ring(MC, 2), Ring(MT, 3)) do
   insertOnRing (MC, 2, Ring(MT, 3), true)
CommitTransactionOn(x) $)
```

traverse the data - possibly much later. But even without slips, the data designer hasn't anticipated remarriage etc. Nor is it easy to understand how to modify the structure design to avoid this deficiency.

Such structures are the precursors of the object types that appear in later models. In this case they appear in CODASYL, with the operations little changed, but more strongly typed, and with a reasonable naming convention. In RSP the persistent data was that reachable from a node given as a parameter to the procedure, *CommitTransactionOn*, that translates data into the preserved form, and returned by the procedure, *OpenDBFile*, which reads a structure from a file performing the inverse translation.

A practical problem arose with parameters in most actual languages and packages. The routine could only take arguments of specific types. Hence a plethora of variants *addRealDataElement, addIntegerDataElement*, etc., were necessary[2]. Results, such as database pointer values, could not be returned from functions as they would not map into the space for scalars. Consequently, awkward mechanisms such as passing vectors as parameters to hold results were necessary. One also has to return an indication of success or failure. Although individually they are not fundamentally difficult problems, the messy detail they introduce into programming made it quite clear that the library of routines approach, and the languages as they existed were quite inadequate.

[2] In untyped languages like BCPL they were necessary to inform the storage system of the data's type and size

No explicit consideration was given to achieving persistence in these libraries. Some ad hoc system which mapped to the filing system's naming conventions sufficed. Only a few had the idea of transactions encapsulated in the library. Possibly there were utilities you could run for recovery if your program or hardware blundered. But the fact that data or new rings could be added to a node or removed at any time, meant that data evolution, necessary with long term persistence, was achieved. (Today, that same effect is achieved with schema editors and reorganisation.)

4.3. Abstract structure libraries

The problem with the data structure library was that it exposed too much detail, particularly detail associated with representing the data on the machine. A simpler model was sought. The most popular was that based on sets. As an example we take Child's Set Theoretic model [Childs 68]. This manipulated sets of tuples via a Fortran call interface. The data retrieval example (test case 3 of figure 3-5) is shown in figure 4-4 after idealising and simplifying the interface, and assuming that the data is modelled in sets similar to the relations in figure 3-2.

Figure 4-4: Example of a set theoretic like program to do the ancestors retrieval

```
      SUBROUTINE PANT (X)
      REAL  PERSON, PC, SOFAR, START

      CALL OPENDB ("EXAMPLE")
      CALL GETSET ("PERSON",PERSON)
      CALL GETSET ("PC",PC)
      CALL NTSET  ("RESULT",SOFAR)
      CALL STRSEL (PC,2,X,START)
      CALL PROJECT(START,1,START)
      IF EMPTY    (START) GOTO 200
1     CALL JOIN   (START,1,PC,2,START)
      CALL PROJECT(START,1,START)
      IF EMPTY    (START) GOTO 100
      CALL UNION  (START,SOFAR,SOFAR)
      GOTO 1
100   CALL JOIN   (START,1,PERSON,1,START)
      CALL PROJECT(START,2,4)
      CALL PRINT  (START)
      CALL DELSET (START)
      STOP
200   .......
```

This is biased towards the relational nomenclature, but is typical of the subroutine library. The existence of many functions to map names (addresses) of one conceptual store to the other has started appearing (the first four calls). Obscure parameters make the interpretation of the program difficult, and leave some of the opportunities for

undetected error of the previous example (the integers identifying tuple elements). A plethora of subroutines are necessary. Consider selection expressions - one wants to do comparisons for, say, 4 base types, with 6 comparators, and either with column and constant or column and column giving 48 routines. But machine efficiency would suggest providing OR and AND on conditions - the complexity of parameters needed then rises alarmingly. Transferring data from a set to the program and performing updates also requires large sets of subroutines. The net result is a very large package - difficult to implement, learn about and use.

Another problem not solved is to enforce consistent interpretation of the data. This can only be achieved by storing its description. This itself requires naming its parts. The result is two independent name spaces - violating the principle of correspondence. In the programs it leads to an ever increaasing number of parameters and calls to perform name mappings. These obscure the program and defeat the programmer.

4.4. Summarising the call interface method

It should be noted that these libraries constitute an exploration of potential higher level data object types. They had undoubted value. But they are fundamentally unsound as a way of presenting persistent data; yet they are still being produced as "data base management systems", particularly on micros today. The problems are a lack of consistent typing and a lack of consistent rules for introducing and using names. Conformity to whatever rules they had depended on the assumption that the programmer wouldn't make mistakes and wouldn't deliberately flout the rules.

The subsequent mechanisms avoid these problems, but in most cases at the expense of flexibility in what data structures may be built and when they may be specified.

5. EMBEDDED DML

Some of the difficulties of using the CALL interfaces described in the last section are the verbosity and organisation of parameters. Another is the lack of checking for consistency between the stored data's structure and the program's behaviour. As a first step, a description of the data must be stored. It is a significant evolutionary step which separates the DBMS from its predecessors.

With the stored data description, many new things become possible. There is a place to associate privacy and monitoring controls with the data. There is a possibility of defining constraints on the data - which is no more than refining its definition. In programming language terms, the type is being refined; a type being a set of allowed

operations and constraints that limit that set. There is a possibility of providing views, and of separating the description of the mapping from program form to stored form into many stages.

But, the experiences before this step have an influence. The data models still look like the data object types manipulated via libraries. There is no reason why the languages for describing the data, and those for manipulating it should become divorced. That they did was a dead-end evolutionary experiment. Although an improvement over the call interfaces, it has led to incompatible and incomprehensible relationships between programs and databases. This is attributable to treating the surface issues rather than their semantic foundation.

For the time being, let us agree that a persistent data description describing the persistent data is mark of progress. How should it be used? It is infeasible to check each subroutine call's parameters against the description for two reasons:

- The applicability of the call depends on its context and the history of operations. These are not easily discovered or recorded.
- The cost of performing the checks at every subroutine call is prohibitive. With some systems, which conduct a moderate subset of the possible checks, there is often 90% of CPU utilisation going on checking, and there are many DBMS that are CPU limited.

So there is an engineering argument for doing something. The obvious alternative is for the compiler to utilise the data description and its knowledge of context and program structure to do as much checking as possible at compile time. But, with the data description and the programming language not being designed as a coherent whole, this is very difficult. Many systems, therefore, adopted an intermediate position. Only some of the potential checking is achieved. A preprocessor processes text which consists of a mixture of program statements and DML statements. The DML statements are identified by some marker. The preprocessor checks them and replaces them by code in the host language, mostly declarations and library routine calls. The advantages are that some checking has been done, and that the programmer has a convenient way of writing the "parameters". Disadvantages are:

- The extra pass in program preparation
- The mixture of two languages
- The obscurity of error reports from the host language compilation and

execution

- The fact that the DBMS can't take advantage of the incomplete set of checks already made.

The last disadvantage is fundamental. A DBMS must protect the data. But a person could write or modify the host language source himself, make errors or deliberately write code which damages irreplaceable stored data. However improbable such an error may be, the risk cannot be taken. The checks have to be made all over again.

We conclude that a safe and efficient DBMS must "know" about programs and keep an authenticated record of compilations of modules, and the assembly of modules into programs. Thus the program construction system must become an integral part of the DBMS. (There are other reasons for doing this; providing better program development environments and improving execution efficiency.)

5.1. An example using an embedded language

Many systems now exist where the preprocessor has been replaced by a compiler which has been extended to process the composite language. This alleviates some of the problems, though the form of the language, in particular the mixing of two independently designed languages, is still apparent. This is illustrated in the example shown in figures 5-1 to 5-6. The Bachmann diagram in figure 3-1 shows the CODASYL data structure assumed in these examples. Figure 5-1 shows the CODASYL schema required for the example (task 1 of figure 3-5). Figure 5-2 shows the appropriate SubSchema for producing the two queries (tasks 2 and 3 of figure 3-5). Figure 5-3 shows the program for task 2, and figures 5-4 and 5-5 show the main program and a necessary subroutine to compute the ancestors (task 3). The final figure in this sequence shows how to record the birth (task 4 of figure 3-5).

The reader is invited to consider the general impression given by these programs. Their size and lack of regular structure makes them difficult to read. (Remember that simple problems were chosen, and that these programs were left incomplete and without many comments.) The description of the data given in the first two figures is difficult to relate to the programs given in the later figures. It is quite hard to see how the algorithm for extracting the ancestors matches the given datastructure. Imagine that you were a maintenance programmer, and hadn't had the benefit of the discussion earlier in the paper. Then you would find it essential to understand the relationship between the data's description and the algorithm.

Figure 5-1: A CODASYL schema to represent the family tree

```
SCHEMA name is PEOPLE.
AREA   GENERATIONS.
AREA   MARRIAGES.

DATABASENAME PNAME; PICTURE A(40).

RECORD type  PERSON;
  WITHIN area  GENERATIONS;
  LOCATION mode is CALC on  NAME USING PNAME;
  DUPLICATES are NOT allowed;
  02  NAME;     PICTURE A(40);
  02  SEX ;     PICTURE A; CHECK "M", "F";
  02  DOB ;     DATE;
  02  DOD ;     DATE.
RECORD type BIRTHLINK;
  WITHIN area  GENERATIONS; LOCATION is VIA  ANTE.
RECORD type  MARRIAGE; LOCATION mode is VIA  PMH;
  WITHIN area  MARRIAGES;
  02  FROM;     DATE;
  02  TO;       DATE.

SET  DESC; OWNER  PERSON; MEMBER  BIRTHLINK;
  MEMBERSHIP is MANDATORY AUTOMATIC;
  SELECTED by CURRENT of OWNER.
SET  ANTE; OWNER  PERSON; MEMBER  BIRTHLINK;
  MEMBERSHIP is MANDATORY AUTOMATIC;
  SELECTED by CALC USING  PNAME.
SET  PMH; OWNER  PERSON;  MEMBER  MARRIAGE;
  MEMBERSHIP is MANDATORY AUTOMATIC;
  SELECTED by CURRENT of OWNER.
SET  PMW; OWNER  PERSON;  MEMBER  MARRIAGE;
  MEMBERSHIP is MANDATORY AUTOMATIC;
  SELECTED by CALC USING  PNAME.
```

Consider now each example in a little more detail. In doing this, we should try to consider predominantly the relationship between language and data base model typified here, and try to surpress our observations on those considered independently. (For a discussion of Fortran see [Feldman 76] and of CODASYL see [Olle 78]). The subschema in figure 5-2 is being required to do two things:

1. Perform the accepted and logical role of a subschema, in identifying a logical subset of the schema, and potentially some view and protection transformations. An example is the omission of the sets and records concerned with marriage.

2. Perform a translation between the names, name formation rules and base types to make those in the database useable in the language. This is not

Figure 5-2: A CODASYL subschema to represent the family tree

```
SUBSCHEMA PERSN, (SCHEMA = "PEOPLE").

ALIAS (AREA ) GNRTN = "GENERATIONS".
ALIAS (AREA ) MRRGS = "MARRIAGES".
ALIAS (RECORD ) PRSN = "PERSON".
ALIAS (RECORD ) BL   = "BIRTHLINK".

REALM GNRTN

RECORD PRSN
  CHARACTER * 40   NAME
  CHARACTER * 1    SEX
  INTEGER   * 4    DOB, DOD
RECORD BL

SET ANTE
SET DESC
END
```

Figure 5-3: A Fortran program to print all the females

```
      PROGRAM FEMMES
#     INVOKE ( SCHEMA = "PEOPLE", SUBSCHEMA = "PERSN" )
#     OPEN   ( AREA =GNRTN, USAGE = RETRIEVAL )
#     FETCH  ( FIRST, AREA = GNRTN, RECORD = PRSN )
      IF ( DBSTAT = 2226 )  GOTO 200
1     IF ( SEX .EQ. "M" )   GOTO 2
      CALL PRNME( NAME )
      CALL PRNDT( DOB )
2     CONTINUE
#     FETCH  ( NEXT, AREA = GNRTN, RECORD = PRSN )
      IF ( DBSTAT .NE. 2207 )GOTO 1
#     FINISH( ALL )
      STOP
200   ...
```

descriptive effort which serves any intrinsically valuable purpose and it introduces a translation which makes the system harder to understand. Examples are the use of the **ALIAS** statement and use of the INTEGER*4 type.

Consider the first program fragment (figure 5-3). It shows some of the surface features of its mixed ancestory. The different typography has been used to draw attention to the fact that the DML sublanguage has a notion of reserved words whereas Fortran does not. It uses keywords as the parameter passing mechanism, whereas Fortran has positional parameters. It has its statements flagged with #. These changes in convention inhibit easy reading and programming. For example, could the second **FETCH** have been labelled? More fundamentally, how does someone reading these programs recognise that

Figure 5-4: A Fortran and CODASYL DML program to print the ancestors

```
      PROGRAM      ANTE
      INTEGER*2    CURGEN( 3, 2, 1000 )
      INTEGER      HWMNY, OLD, NEW
      CHARACTER*40 WANT
      DATA         WANT/"Janner            Curnow            "/
#     INVOKE ( SCHEMA = "PEOPLE", SUBSCHEMA = "PERSN" )
#     OPEN ( AREA = GNRTN, USAGE = RETRIEVAL )
      CALL COPY( WANT, NAME )
#     FIND ( RECORD = PRSN )
      ...
      HWMNY = 1
      OLD   = 1
      NEW   = 2
#     ACCEPT ( CURRENCY = CURGEN( 1, OLD, 1 ), RECORD = PRSN )
      CALL PRNME( WANT )
      WRITE( 1, 1001 )
1     CALL NEXGEN( CURGEN, HWMNY, OLD, NEW )
      CALL SWAP( OLD, NEW )
      IF( HWMNY .NE. 0 ) GOTO 1
#     FINISH ( ALL )
      STOP
      ...
1001  FORMAT("'s antecedents are :- "/)
      END
```

Figure 5-5: A Fortran subroutine used by the program in figure 5-4

```
      SUBROUTINE NEXGEN( DBKYS, COUNT, IN, OUT )
C Given a generation of COUNT people in DBKYS(*, IN, * ) it puts the next older
C generation in DBKYS(*, OUT, * ) leaving the size of that generation in COUNT.
      INTEGER*2  DBKYS( 3, 2, 1000 )
      INTEGER    COUNT, IN, OUT
      INTEGER    INCNT
#     INVOKE ( SCHEMA = "PEOPLE", SUBSCHEMA = "PERSN" )
      INCNT = COUNT
      COUNT = 0
      DO 10 I = 1, INCNT
#     FIND ( DBKEY = DBKYS( 1, IN, I ))
#     IF ( EMPTY, SET = ANTE ) GOTO 10
#     FIND ( FIRST , RECORD = BL, SET = ANTE )
1     CONTINUE
#     FETCH ( OWNER , SET = DESC, SUPPRESSING = ANTE )
      COUNT = COUNT + 1
#     ACCEPT (CURRENCY = DBKYS (1, OUT, COUNT), RECORD = PERSN)
      CALL PRNME( NAME )
      CALL PRNDT( DOB )
#     FIND ( NEXT, RECORD = BL, SET = ANTE )
      IF( DBSTAT .NE. 0307 ) GOTO 1
10    CONTINUE
      RETURN
      END
```

NAME has been properly introduced and initialised. For those not familiar with

Figure 5-6: A Fortran and CODASYL DML program to record a birth

```
      PROGRAM ADDMC
      CHARACTER*40 JC, MT, MC
      DATA JC/"Janner     Curnow   "/,
     *     MT/"Moyanna    Trefusis "/, MC/"Morwenna    Curnow    "/
#     INVOKE ( SCHEMA = "PEOPLE", SUBSCHEMA = "PERSN" )
#     OPEN ( AREA = "GNRTN", USAGE = EXCLUSIVE )
      CALL COPY( JC, NAME )
#     FIND ( ANY, RECORD = PRSN )
      IF( DBSTAT .NE. 0 )                GOTO 100
      CALL COPY( MT, NAME )
#     FIND( ANY, RECORD = PRSN )
      IF( DBSTAT .NE. 0 )                GOTO 101
      CALL COPY( "F", SEX )
      DOB = TODAY()
      DOD = 0
#     STORE ( RECORD = PRSN )
      CALL COPY( JC, NAME )
#     FIND ( ANY, RECORD = PRSN, SUPPRESS = ANTE )
#     STORE ( RECORD = BL )
      CALL COPY( MT, NAME )
#     FIND ( ANY, RECORD = PRSN, SUPPRESS = ANTE )
#     STORE ( RECORD = BL )
#     FINISH ( ALL )
      STOP
      ...
```

CODASYL, DBSTAT, and funny numbers such as 2207, are used to inform the program of the results of the last action. The peculiarity of the mechanism is a direct consequence of the mismatch between language and database.

Taking another look at the program to list the antecedents (figures 5-4 and 5-5), we can see the mismatch becoming more serious. The absence of constructed types in Fortran like **RECORD** and **SET** means that it is hard for even a good programmer to show the relationship between his algorithm and the data structure in the database. But lack of a good set facility in the language has led the programmer to code an algorithm which is also **wrong** in at least two respects:

1. It searches the family tree by constructing sets of people corresponding to each generation in the array CURGEN, alternate generations having the second index 1 or 2. One only needs to go back 13 generations, or a little over 200 years for humans to get an overflow of this representation. If that happens, one suspects that the maintenance programmer would patch the program so it checked COUNT didn't become greater than 1000, and/or increase the size of CURGEN. This wouldn't solve the problem as if we go back to the time of Christ, each person would have $\sim 2^{100}$ ancestors printed.

2. The "sets" weren't in fact sets, so our apocryphal programmer, checks a person isn't in CURGEN before putting him in. But that doesn't take care of the problem either as people may have common ancestors different distances up the tree on different branches.

The conclusion is that this program is basically wrong. Yet it isn't apparent, from the program or the data, and the problem is unlikely to show up on test data. When it does, it will be difficult for the maintenance programmer to make the fundamental changes necessary, because he will not have presented to him a description of the data and algorithm which makes clear what is happening. Yet this is a version of a common problem, described after being simplified to the point of triviality.

As a final demonstration, note that the update program in figure 5-6 inserts the two BL records it creates into two sets each. You certainly can't deduce that from the program, and it would be quite hard given all the information in Figures 5-1 through 5-6 to convince yourself that the correct parentage is recorded. Take up the challenge now, before you read on.

Most database programming is done in contexts similar to this: PL1 + SQL, COBOL + CODASYL, COBOL + FORTRAN etc. It is little wonder that it is error prone and expensive. However, the investment in existing databases and programming languages is so significant that research into improving this interface is undoubtably worthwhile. Buneman has developed a technique that is appropriate for strongly typed languages such as Pascal and Ada [Buneman *et al.* 82] and continues to develop interfaces, for example using the Zermelo-Frankel notations of KRC [Buneman & Davidson 83].

6. INTEGRATING PROGRAMMING LANGUAGES

Dissatisfied with just glueing predefined languages together, researchers have attempted to extend programming languages with database facilities, in a way which is consistent with the original design philosophy of the parent language. It is difficult to delimit precisely where this group takes over from embedding languages, and it is difficult to decide where such languages should be treated as totally new languages. We consider here the languages which look like extensions of Pascal with relations: Pascal/R [Schmidt 77, Schmidt 78], PLAIN [Wasserman *et al.* 81], RIGEL [Rowe & Shoens 79], THESEUS [Shopiro 79], and a language which looks like an extension to ADA: ADAPLEX [Computer Corporation of America 83].

6.1. Pascal/R: a composition of Pascal and relations

Figure 6-1: Pascal/R datastructure declaration for a family tree

```
Type Pnum       = integer;
  Date          = Record day:1..31; month:(Jan, Feb, ...); year:integer end;
  PersonRec     = Record pn:Pnum; name:string; sex:(M, F); dob, dod:Date end;
  BirthRec      = Record parent, child: Pnum end;

  PersonRel     = Relation <pn> Of PersonRec;
  BirthRel      = Relation <parent, child> Of BirthRec;
Var People      : DataBase person: PersonRel; pc: BirthRel end;
```

For tutorial purposes, the language Pascal/R is considered further. We choose it because it is both the earliest, and simplest of these languages and in many respects the best. It has been implemented on the DEC System 10 computers and used both for research and for large scale commercial applications. A sucsessor to it, DBPL is under construction for the DEC VAX computers [Mall *et al.* 83] based on Modula-2 [Wirth 83].

Figure 6-1 shows a data structure which would represent our standard example (task 1 in figure 3-5). Compare this description of the data structure with that given in a relational description language figure 3-2. You will notice that the end effect is very similar, except that the types of the components have been identified and named as well as the instances. This is an advantage, in the sense that new variables could be introduced for use in the program with little extra effort. The notation **Relation** constructs a relation type from the record type given, and at the same time gives a list of fields of that relation which will serve as a unique identifier. The construct **DataBase** is used here to construct a new variable in the program, whose value is a database of two relations, PERSON and PC. We note in passing that this is a very general mechanism for describing a database, and that in principle the language could allow any datatype within the **DataBase** construct. If it did, then the provision of persistence in Pascal/R would be properly orthogonal, and the **DataBase** construct would serve to specify the name, type and naming system of data to have persistence. Probably for engineering reasons and lack of resources, the Pascal/R team have only implemented this construct for fields of type relation. They have also restricted the concept, allowing only one such variable per program, denying the programmer the chance of writing programs that operate between databases. They have not permitted it to occur within a procedure, so that it is not possible to provide a procedure library for use with a database without requiring that the user learn unnecessary details about the database. The allowed fields within a record that is to be formed into a relation are also restricted to simple types. The system for naming and describing them, however, is entirely consistent with the other notations of Pascal and can be used quite generally throughout the program.

Figure 6-2: Pascal/R program to print the females in the family tree.

```
Program femmes( People, Output );
Type Pnum     = integer;
  Date        = Record day: 1..31; month: (Jan, Feb, ... ); year: integer end;
  PersonRec   = Record pn: Pnum; name: string; sex: (M, F); dob, dod: Date end;
  BirthRec    = Record parent, child: Pnum end;
  PersonRel   = Relation <pn> Of PersonRec;
  BirthRel    = Relation <parent, child> Of BirthRec;
Var People    : DataBase person:PersonRel; pc:BirthRel end;
....                {declaration of WriteDate and WriteName}
Begin
  With People Do
    Foreach p In person Do
      If p.sex = F Then
        Begin WriteName( p.name ); WriteDate( p.dob ) end;
end.
```

Figure 6-2 shows such a Pascal/R program using these data types to print the list of females (task 2 of figure 3-5).

Note that in this program, the existing database is bound to the variable describing it via a program parameter. Presumably, the system verifies that the description given in the program and the description stored with the data are compatible, before executing the program. This could in principle achieve the effects of both the subschema and the **INVOKE** operation shown in the last set of examples. Pascal/R does not utilise this opportunity, nor does it allow programs to operate on more than one database. The iterator **Foreach** has been introduced to solve one of the problems that must be solved in any language that introduces large scale types, such as relations, into a language which hasn't had them before. The problem is to make a transition from a group of objects to an individual, without introducing a complicated selection rule. Schmidt chose to introduce the new type as an extension of the idea of **set** [Schmidt 78], and the **Foreach** operator then implements the axiom of choice. We note in passing that it is consistent with the iterator suggested by Tennent [Tennent 77] for other Pascal data types, such as arrays. Actually Schmidt hedges his bets a little, as he also provides an interface rather like Pascal's operators on **File** structures. The program to enumerate the females is simple to read, and can also be expressed using more of the relational expressions provided by Pascal/R. A revised and abbreviated version which does this is shown in figure 6-3.

The transformation of the former program into this program is straightforward, but the effects of doing this are significant. For example, it has been possible to give the subset to be printed a meaningful name, and the method of constructing it is similar to that of the relational calculus query languages. These have an advantage of being

Figure 6-3: Another Pascal/R program to print the females.

```
Program females( People, Output );
Type Pnum  = integer; ... {The remaining type declarations}
Var  People: DataBase person:PersonRel; pc:BirthRel end; females:PersonRel;
...          {declaration of WriteDate and WriteName}
Begin
  With People Do
    Begin
    females := [Each p In person: p.sex = F];
    Foreach p In females Do
      Begin WriteName( p.name ); WriteDate( p.dob ) end end
end.
```

concise, and allowing optimisation and sensible use of access structures.

The program in figure 6-4 demonstrates how Pascal/R might print the ancestors (task 3 figure 3-5).

Figure 6-4: A Pascal/R program to print the ancestors of Janner Curnow

```
Program Ancestors( People, Output );
...   {Declarations of types and People as in figure 6-2}
Var Temp, Generation, Ancestors:PersonRel; p:PersonRec;

Procedure NextGen(OldGen: PersonRel; Var NewGen: PersonRel);
{given a generation in OldGen, it calculates the next in NewGen}
  With People Do
    NewGen := [Each q In Person: Some b In pc: (q.pn = b.parent and
                                 Some p In OldGen: p.pn = b.child) ];
Begin
  With People Do
    Begin
    Ancestors  := []; Generation := [Each p In Person: p.name='Janner Curnow'];
    While Not empty( Generation ) Do
      Begin
      NextGen( Generation, Temp ); Ancestors :+ Temp; Generation := Temp end;
    writeln( 'The  ancestors of Janner Curnow are: ');
    Foreach p In Ancestors Do
      Begin WriteName( p.name ); WriteDate( p.dob ) end end
end.
```

This program illustrates the value of being able to manipulate the same types of data when they are temporary as when they are persistent. The introduction of the ANCESTORS, GENERATION, and TEMP variables and their manipulation by assignment and set union (:+) illustrate this. For the first time in our examples, the result does not contain errors, as the variable ANCESTORS is treated as a set. The slightly revised version of the tree scanning loop shown in figure using, in addition, set difference (:-) offers some improvements. If the data happened to erroneously contain cycles, this program would still converge. If it contains people who are ancestors via more than one

Figure 6-5: A revised version of the tree searching loop

```
begin NextGen (Generation, Temp); Temp :- Ancestors; Generation := Temp;
  Ancestors :+ Temp; end
```

path, these paths being of different length, it will only search their ancestors once, whereas the earlier version would search once for each path by which they were reached. For deep trees, such as the data structures that occur in parts explosions, and for animal and plant breeding data where inbreeding is deliberate, the change should be a significant improvement. The simplicity with which the programmer can write such operations is impressive. It is likely to enable not only more readable and correct programs to be written, but also to make feasible global optimisations such as this last modification. It is a symptom of the difficulty of being completely consistent when extending a language, that we could not write:

```
Generation := NextGen(Generation);
```

and hence avoid the introduction of the variable TEMP which does not contribute to the clarity or efficiency of the program. It is a failure of the design principle of *data type completeness*, which is inherited from the parent language Pascal. However, further failures have been introduced by the Pascal/R team: the restriction on fields in relation records and database variables, for example. These would actually prevent one from having fields of type DATE in the PERSON records as we have.

For those who prefer the more explicit specification of the next generation the procedure NEXTGEN could have been written as shown in figure 6-6.

Figure 6-6: Another rendition of a procedure to compute the next generation

```
Program Ancestors2( People, Output );
...  {Declarations of types and People as in figure 6-2}
   Temp, Generation, Ancestors: PersonRel;
Procedure NextGen( OldGen: PersonRel; Var NewGen: PersonRel );
{given a generation in OldGen, it calculates the next in NewGen}
  With People Do
    Begin
    NewGen := [];                          {An empty set of Ancestors}
    Foreach p In OldGen Do
      Foreach b In pc Do
        Foreach q In Person Do
          If q.pn = b.parent And b.child = p.pn Then NewGen +: [q];
    end;
...                                        {The rest of the program}
```

It is interesting to note, that not only is the programmer free to express the algorithm in this and many other possible ways, but that the Pascal/R team have managed the considerable engineering feat of detecting most of the cases where such iteration can be transformed into an expression which they can safely optimise [Jarke & Koch 82, Jarke

& Koch 83]. This is important as it allows the programmer a necessary freedom.

These examples illustrate a conciseness, and a consistent and complete description of the program and data, which is a major step forward. Similar examples might have been constructed in PLAIN, RIGEL or THESEUS.

Finally consider the example in figure 6-7, which shows how a new birth might be encoded.

Figure 6-7: Pascal/R program to record a birth

```
Program RecordBirth(Input, Output, People);
Type Pnum       = integer;
  ID            = Array [1..5] Of char;                 {NOTE}
  PosInt        = 0..1000000;                           {NOTE}
  Date          = Record day:1..31; month:(Jan, Feb, ...); year:integer end;
  PersonRec     = Record pn:Pnum; name:string; sex:(M, F); dob, dod:Date end;
  BirthRec      = Record parent, child: Pnum end;
  UIDRec        = Record nme:ID;                        {NOTE}
                         value:PosInt End;              {NOTE}
  PersonRel     = Relation <pn> Of PersonRec;
  BirthRel      = Relation <parent, child> Of BirthRec;
  UIDRel        = Relation <nme> Of UIDRec;             {NOTE}
Var People      : DataBase person: PersonRel; pc: BirthRel;
  UIDs: UIDRel end;                                     {NOTE}
  JCrel, MTrel: PersonRel; JCrec, MTrec: PersonRec; JCpn, MTpn, Nextpn: Pnum;
  PNrel: UIDRel; PNrec: UIDrec;                         {NOTE}
  p: PersonRec;
Begin
With People Do
  Begin
  JCrel := [Each p In PersonRel: p.name = 'Janner Curnow'];
  this(JCRel, JCrec); JCpn := JCrec.pn;
  MTrel := [Each p In PersonRel: p.name = 'Moyana Trefusis'];
  this(MTRel, MTrec); MTpn := MTrec.pn;
...             {Check consistancy constraints here}
  {Get last used pn, record one to be used}             {NOTE}
  PNrec := UIDs['PN    '];                              {NOTE}
  Nextpn := PNrec.value;                                {NOTE}
  Nextpn := Nextpn + 1;                                 {NOTE}
  UIDs :& [<'PN    ', Nextpn>];                         {NOTE}
                {Create and insert a record for the new baby}
  person :+ [<Nextpn, 'Morwenna Trefusis', F, <22, Sep, 82>, Alive>];
                {Create links to parents and add those to the births relation}
  pc :+ [<JCpn, Nextpn>, <MTpn, Nextpn>]; end
end.
```

This program does make clear how the birth is being recorded. However there are some infelicities.

For example, one would like the value of the last used identifier to be held in PEOPLE in the following fashion:

```
Var People: DataBase person: PersonRel; pc: BirthRel; Nextpn: integer end;
```

So that it may be used directly, and incremented directly, rather than via the circumlocutions highlighted in the program with *NOTE* comments. Similarly, it would be desirable to be able to keep with the database definition as constant fields of PEOPLE, the values to be used for ALIVE or any other such constants. Again we see the importance of the principle of data type completeness. But at least Pascal/R improves on Pascal in this respect by introducing a notation for the literals of more of its datatypes; that is for records and relations. As Berry has remarked [Berry 81], this deficiency of Pascal was not shared by Algol68 [van Wijngaarden *et al.* 69].

6.1.1. Summary of Pascal/R

Within the confines already established by Pascal, Pascal/R achieves a consistent treatment of type and of name spaces, so that the program and database have an obvious relationship. The introduction of the new iterators and the relational calculus expressions has significantly enhanced the power of the language for expressing typical database operations. There is also a good interactive facility called DIALOG, which enables quick ad hoc examination of the database. We consider this of importance to the programmer. The two deficiencies that Pascal/R inherits from the Pascal philosophy are a lack of data type completeness (for example only relations may be fields of a **database** variable, only one such variable may occur per program and it may not occur within a procedure) and the datatypes available as domains in a relation are restricted to *some* of the Pascal types.

6.2. The DAPLEX extension to ADA

The ADA extension, ADAPLEX, being developed at the Computer Corporation of America (CCA), in Boston, brings in new ideas. The major impact is an attempt to provide a much richer set of datatypes for modelling the persistent data. These are the datatypes of the functional data model according to DAPLEX [Shipman 81] and include the notions of sets, of any entity being a value, of functions whose results are single entities (*Single valued functions*), of functions whose results are sets (*Multi-valued functions*), and of type hierachies and property inheritance. More has already been said about this data model in other chapters, so we concentrate here on the combination of that model with ADA.

The task of combining a new concept with ADA is not straightforward, as there is a limited understanding of how to implement ADA, and the language is already complex.

There are restrictions which prevented CCA from altering the language ADA in any way, and the project was undertaken in combination with an ambitious project, Multibase [Smith *et al.* 80a, Smith *et al.* 80b], to build a heterogeneous distributed database system. Thus we look here at only one aspect of a very large scale project. To do this, we consider first how DAPLEX is introduced into ADA.

Instead of using the existing record types in ADA, a new type ENTITY is introduced, with a notation similar to that for records, rather than the notation used by DAPLEX. Any ADA module that operates on the persistent data manipulates these entities, and the values within them. Preparatory to doing this, it must declare the entity types it will use, and the relationships, among their types. The idea of a set is introduced in a limited way. Sets would be generally useful, but they can only appear in the context of an **atomic** statement, and types constructed using **set of** cannot be passed as parameters, or as results, and cannot appear in arrays or records. Though understandable because of the constraints on CCA, we find this lack of data type completeness irritating. Even more irritating is the fact that one cannot write generic operations, such as union, over sets.

Figure 6-8: The data structure definition in ADAPLEX for the family tree

```
database Genealogy is
  type GENDER is (M,F);
  type MONTH  is (JAN, FEB, MAR ...);
  type PERSON;                          --Needed for recursive definition

  type PERSON is entity
    Name : STRING(1..20);
    Sex  : GENDER;
    Dob_day : INTEGER range 1..31;
    Dob_month : MONTH;
    Dob_year: INTEGER;
    Mother : PERSON withnull;
    Father : PERSON withnull;
    Children  : set of PERSON;
  end entity;
  unique Name within PERSON;
end Genealogy
```

We should now look at a few examples of how we believe ADAPLEX programs may look when the language is available. The first figure (6-8) accomplishes the task of describing the data structure to represent parentage (task 1 of figure 3-5). Note that a database as declared in this example is similar to an ADA module. This is a promising approach as, if taken to its logical conclusion, it will introduce names for database components in exactly the same way as they are introduced for program components.

Another language construct in ADA, which they might have made a database equivalent to, is the task. This might have been sensible, as many tasks require concurrent access to a database, which would match with ADA's inter-task structure. As in Pascal/R, the allowed values within an **entity** are restricted, here to scalar types, strings, other entities, and sets of these. In particular, dates once again could not be modelled with their structure, and individual fields associated by naming convention have been used. It seems strange to have to think of the elements like SEX, MOTHER, etc., as functions when they look like traditional fields, but the difference shows only in the way the semantics are organised and explained. Thus the fields NAME through FATHER introduce single valued functions (Shipman's single arrow →), and the field CHILDREN introduces a multivalued function. The **withnull** construct allows those fields an undefined value when the parents are not known.

Figure 6-9: ADAPLEX program to list the females

```
with Genealogy;
use Genealogy;
Print_Females: atomic
   for each P in PERSON where Sex(P) = F by Name(P) loop
      PUT(Name(P)); PUT(Dob_day(P)); PUT("/"); PUT(Dob_month(P));
      PUT("/"); PUT(Dob_year(P)); NEW_LINE;
   end loop;
end atomic;
```

Given this data structure, the program shown in figure 6-9 will print the list of females. The **extent** attribute of an entity class name gives access to a set of all the instances of that entity class, however, it is implicit after **in** in the **for each** statement. **for each** has a similar role to that which it plays in Pascal/R, providing iteration over the set. The **by** construct here allows the programmer to specify a sort order in which he wants the elements of the set presented. This is of practical value, and is neatly incorporated into the **loop** statement. The **atomic** construct is also new. It performs two roles: it identifies the parts of programs where the new constructs introduced as accessories to standard ADA may appear. It also, when updates are coded, delimits the units of transaction. This sort of construct is obviously necessary in any complete database language. It allows the runtime system to organise concurrency control, as any statement which references the database must be within an **atomic** statement, and it allows commitment or rollback of transactions to be organised.

The program shown as figure 6-10 shows how the ancestors may be printed (task 3 figure 3-5). Three set variables *Genertn*, *NextGen*, and *Ancestors* are introduced, to hold the generation currently being considered, the parents of that generation, and the total set of ancestors located so far, respectively. The algorithm used is similar to the

Figure 6-10: ADAPLEX program to print the ancestors

```
with Genealogy;
use Genealogy;
Compute_Ancestors:
 declare
  Genertn, NextGen, Ancestors : set of PERSON;          -- Initialized to empty
 atomic
  include (P in PERSON'EXTENT where Name(P) = "Janner Curnow") into Genertn;
  while Genertn is not empty loop
    include Genertn into Ancestors;
    include Mother(Genertn) into NextGen;
    include Father(Genertn) into NextGen;
    exclude Genertn from Genertn;                       --empty the set;
    include NextGen into Genertn;
    exclude NextGen from NextGen;                       --ditto
  end loop;
  for each P in Ancestors loop
    PUT(Name(P)); PUT(Dob_day(P)); PUT(Dob_month(P)); ...; NEW_LINE;
  end loop;
end atomic;
```

basic one used for Pascal/R (figure 6-4), but it was necessary to flatten the recursion into an iteration as it does not appear possible to pass the sets as parameters of procedures. This is an alarming shortfall from the *principle of data type completeness*, and though the consequences aren't serious here, this deficiency could result in very unstructured and obscure code. By a similar omission, it was not possible to simply assign new values to set variables except by the device of arranging that the destination set be empty, and then by **including** the new set **into** it, which performs a set union. It seems that the designers of the language did not accept that people would want to program with the data types they were allowed to store in the database. Similarly we would have preferred to write

```
Genertn := {}
```

rather than the less obvious

```
exclude Genertn from Genertn
```

but this is explicitly dissallowed in the reference manual [Computer Corporation of America 83] page 14 of section 2. There are two reasons given for this omission. The ADA philosophy frowns on criptic symbols such as :+ for set union. That problem can be overcome by using words such as **becomes** and **without**. The assignment statement is considered dangerous, as in

```
PERSON'EXTENT becomes empty
```

which might delete the whole of the database. This argument is false, as there will always be many ways in which a programmer can destroy a database in any language

which is powerful enough to be useful; making the language more inconvenient only increases the chance of doing it accidently.

Figure 6-11: ADAPLEX program to record a birth

```
with Genealogy;
use Genealogy;
Add_Person:
 declare
  Child: PERSON;
atomic
  Child := new PERSON
     (Father ⇒ {P in PERSON where Name(P) = "Janner Curnow"},
      Mother ⇒ {P in PERSON where Name(P) = "Moyana Trefusis"},
      Name ⇒ "Morwenna Trefusis",
      Dob_day ⇒ 22, Dob_month ⇒ SEP, Dob_year ⇒ 1982,
      Sex ⇒ F, Children ⇒{});
  include Child into Children(Father(Child));
  include Child into Children(Mother(Child));
exception
   when NULL_ENTITY =>
        PUT("Can't find parents");
   when UNIQUENESS_CONSTRAINT =>
        PUT("Duplicate child's name");
   end;
end atomic;
```

The figure 6-11 illustrates how to program the update task. Again we see set selections from the extent of a class. The example program creates a new entity of type person which will be automatically included in the extent of person. It has to be explicitly included in the sets representing the children of each parent. It uses the exception mechanism to abhort the transaction if the data is inconsistent.

6.2.1. Summary of ADAPLEX

Both these integrated languages (Pascal/R and Adaplex) have much in common. They both show a very considerable improvement in the clarity of programs, and in the power available from quite concise notations. However, the ADAPLEX endeavour seems to suffer from even less consistent treatment of data types than Pascal/R. This is unfortunate, as basing the data model on functions, which are already a concept in the programming languages, should have led to a consistent treatment of the persistent and transient data. It is not the ideas of DAPLEX that lead to this shortcoming, but the poor treatment of functions in ADA. As *functions* are not even permitted as parameters in ADA, let alone treated as first class data objects, it is not surprising that ADAPLEX has had to construct an independent set of concepts. The maintenance rules of ADA then prevent those concepts from pervading the whole language. It *is* surprising that the

ADA design team [Ichbiah *et al.* 79] did not consider the issue of how to program with persistent data, since the use of stored data is so pervasive. CCA in tackling this issue have taken up the important challenge of trying to repair this shortcoming, a step which requires both vision and courage. So, one must admire ADAPLEX as an engineering feat. It is not through lack of design effort that ADAPLEX is as it is. The majority of decisions in its design are justified, given our current understanding of how to implement persistence, by the need to engineer a system that will perform adequately with large distributed bodies of data. At present, the implementation effort is concentrated on an interactive subset, together with Multibase and a database handler in ADA.

At least one other attempt is being made to construct a persistent programming facility for ADA [Horowitz & Kemper 83], in this case by the addition of relations.

7. PERSISTENT PROGRAMMING LANGUAGES

In a series of experiments, the persistent programming group at the universities of Edinburgh and St Andrews in Scotland have been trying to develop a consistent treatment of persistence [Atkinson 78, Atkinson *et al.* 81, Atkinson *et al.* 83a, Atkinson *et al.* 83b, Atkinson *et al.* 83c, Cockshott *et al.* 83]. Their motives include:

- An attempt to demonstrate the value of giving high priority to the principle of data type completeness, particularly as applied to a data item's right to persist, which is defined to be independent of data type.
- An attempt to demonstrate that the *traditional* data model based on types which are easy to engineer in a computer, (ie records, arrays and references) can form an adequate data model for data outside as well as within programs.
- An attempt to demonstrate that functions can be first class data objects in persistent languages with assignment, and that such functions can provide many of the traditional database mechanisms.

As a consequence of a rather different target, no new, higher level (bulk) data types are introduced into the language directly. As the following examples show, this has the advantage of flexibility and the commensurate cost of a requirement for rather detailed programming.

7.0.1. Other work on providing persistence

A similar project, adding persistence to ML, was undertaken at Pisa as the language ELLE [Albano 82]. ELLE has developed into Galileo, discussed in the next section.

APL [Falkoff & Iverson 73] and LISP (or rather derivatives like INTERLISP [Teitelman 75]) have had a form of orthogonal persistent data for some time. This form is in fact used by other applicative languages. It consists of a mechanism to save and restore the current workspace. The shortcomings of this approach we have related elsewhere [Cockshott *et al.* 83]. Problems of scale, lack of a transaction mechanism, and lack of adequate mechanism for the independent development of program and data are the chief problems. We note that it is in the mechanism of saving and restoring the workspace, usually to and from some specified file, that the purely applicative language proponents actually implement the updates we consider essential, ie they do it manually with the aid of the operating system.

Recently others have considered applying the approach to languages with assignment in various ways. Brooker [Brooker 82] has proposed an extension of BCPL. This introduces a particular data model through a procedural interface. Another persistent extension of BCPL has been implemented in Cambridge [Gregains & Wiseman 84]. Here the implementation is limited by the low level nature of BCPL. Consequently the programmer still has much responsibility for the correctness of structure representation, which must surely be especially unsafe with persistent data. Similarly it is not possible to operate with more than one database at once. Another attempt at designing a new language with orthogonal persistence is given in [Cormack 83]. This uses an explicit specification of the persistence of objects. Hall has suggested an extension to ADA to provide orthogonal persistence [Hall 83].

7.1. PS-algol applied to the family tree

In presenting PS-algol, the power of functions to provide a view of the initial definition will be presented first, then the update problem and finally the query tasks will be coded.

7.1.1. Data definition in PS-algol

The program in figure 7-1 shows how a comprehensive data structure for the family data might be defined. The **structure** construct defines a record class. The **pntr** construct defines a type which ranges over tokens for all the existing instances of those classes and the special token **nil**. An asterisk preceding a type denotes the type which is all vectors of that type. Note that pointers are not constrained to have members of a particular class as a referend, so that the programmer has had to use comments to make his intentions clear. The structure class *population* has been introduced to demonstrate that sophisticated index structures may be built. The definition is not explicitly transferred to a database by a programmer, but as a programmer uses data in programs,

Figure 7-1: A PS-algol data structure for the family trees

```
structure date(int day,month,year)
structure person(
   int pn;              !guaranteed unique
   string name;         !persons name
   bool sex;            !females are true
   pntr dob,            !to date (of birth)
        dod,            !to date (of death)
        mother,         !to a person record
        father;         !to a person record
   *pntr kids,          !to person records
        spouses)        !to marriage records
structure marriage(
   pntr husband,        !to a person record
        wife,           !to a person record
        began,          !to a date record
        ended)          !to a date record or nil
structure population(
   pntr TbyN,           !to a table person records indexed on pn values
        TbyNme,         !pop. by lastname concatennated with decade
        TbySnd)         !same population indexed on sound of last name
!Assume a number of independent populations accessed from the main table.
```

the corresponding definitions migrate to the database, so that a description of all the data in the database is accreted there.

Figure 7-2: A PS-algol program to set up a new empty population

```
structure population(pntr TbyN, TbyNme, TbySnd)
let people = open.database("people", "Lunga", "write")
if people is error.record do {grumble(); abort}
let needName := true
while needName do
  {write "'nSupply population name"; let popNme = read.a.line()
   let old = s.lookup(popNme, people)
   if old ≠ nil then write "'nName in use [", popNme,"] try another"
   else needName := false}
let newPop = population(table(), table(), table())
s.enter(popNme, people, newPop); commit()      !preserve the population structure
```

It is obviously possible for such a structure to be used in an undisciplined way, but by using the properties of algol scopes and functions as data objects it is possible for one programmer to provide a set of procedures which are the only operations on the data available to the less trusted users. These procedures perform the role of views, but the transformations are such that they are better thought of as an abstract data type. Their use may make it feasible to enforce consistency constraints or any other policy as well as present a more convenient view. Figure 7-2 shows a fragment of a program building

such an abstract type. It is a program to set up an empty population. The first time it is run on a new database, the database will acquire the description of the *population* class as it saves the first instance of the class. The final statement ensures that instance's persistence, as it enters a reference to it in the root table, and all data reachable from that table is preserved with this database. The fact the programmer is working with a database is not obtrusive in these examples.

Figure 7-3 is a fragment of a program which sets up and stores in the database the functions on people. The function to enter a new person is the only one shown. Note that as it is defined, it refers to other items in its static lexical environment, and in order that the function will execute properly when it is subsequently retrieved from a database, the system has to ensure all these are stored too. This results in all the subsidiary functions and all the type definitions it references being stored. Note too that it refers to the variable *lastPN* in which it records a counter used as a source of identifiers for people. This is declared and incremented just like any other integer. Contrast this with the arrangements necessary in Pascal/R (Figure 6-7). Similarly, constants, such as *alive*, *NoKid*, etc., have been recorded in this way. Thus all the associated data and description is stored together and the arrangements for its storage are implicit. But the programmer has left subsequent users only with access to the *MkPerson* procedure (and other similar procedures, we imagine) so that all these other facts are suitably hidden and protected. For example, it is not possible for someone to decrement *lastPN*. This is a powerful view and protection mechanism; it also provides a program and procedure library, and linking mechanism without introducing extra concepts. This is an immediate and satisfactory consequence of making procedures first class data objects in the database. The procedure defined ensures the family tree is a lattice (no cycles) and that the births are biologically feasible, and that all people are in all three index structures. Note that two of these have been tailored to hold duplicates.

7.1.2. Building views on views

The preceding example produced an abstract data type to create people and insert them into the family tree in one operation. This ensured integrity in the sense that indexes and pointers would be consistent. It required the programmer to identify the parents by pointer values. Many programmers, we imagine, would have to deal with identifying the parents by name. Further, since the identification of parents may be critical to correctness, we may wish to impose a consistent procedure for resolving ambiguities and for persuading the user to actually provide this data. The program fragment in figure 7-4 shows such a refined view being constructed in terms of the preceding one. Note how a new operation to set the current population has been introduced, to accomodate programmers who will work in the context of one population

Figure 7-3: PS-algol program to store a new person procedure in the database

```
structure population(pntr TbyN, TbyNme, TbySnd)
structure date(int day, month, year)
structure person(int pn; string name; bool sex; pntr dob, mother, father;
   *pntr kids, spouses)
...
let keyNme = proc(string Anm; pntr Adob → string)
  {let decade = substring(iformat(Adob(year)), 2, 2)
   let last.name = after.last.space(Anm); decade ++ last.name}
let keySnd = proc(string Anm → string); soundex(after.last.space(Anm))
structure same.key(pntr a.persn, another.persn)
let MultiEnt = proc(string key; pntr tab, val)
   {let so.far = s.lookup(key, Apop(Tab)); let new.entry := val
   if so.far ≠ nil do new.entry := same.key(val, so.far)
   s.enter(key, tab, new. entry)}
let NoKid = vector 1::1 of nil; let single = NoKid; let alive = nil
let kid.of = proc(pntr parent, child)
   {if parent ≠ nil do                    !parent known?
      if parent(kids) = NoKid then        !first kid
         parent(kids) := @1 of pntr [child]
      else
       { let kidNo = 1 + upb(parent(kids)); let had.kids = parent(kids)
         parent(kids) := vector 1::kidNo of child
         for i = 1 to kidNo do parent(kids, i) := had.kids(i)}}
...
let is.person = proc(pntr X → bool); if X = nil then true else X is person
let possible = proc(pntr X, Y, the.dob → bool)
   is.person(X) and is.person(Y) and opposite.sex(X, Y) and
   compatible.age(X, the.dob) and compatible.age(Y, the.dob)
let db=open.database("people", "Lunga", "write");
if db is error.record do {grumble(); abort}
let lastPN := 0                           !record of last used person number
let MkPerson = proc(pntr Apop; string Aname; bool Asex; pntr Apa, Ama, Adob)
  {if not possible(Ama, Apa, Adob) do {write "'nParents unacceptable"; abort}
   lastPN := lastPN + 1                   !ensure no two people have same PN
   let ThePrsn = person(lastPN, Aname, Asex, Adob, alive, Apa, Ama, NoKid, single)
   i.enter(lastPN, Apop(TbyN), ThePrsn)
   let key = keyNme(Aname, Adob)          !now compute index on name & DOB
   MultiEnt(key, Apop(TbyNme), ThePrsn)
   MultiEnt(keySnd(Aname), Apop(TbySnd), ThePrsn)
   kid.of(Ama, ThePrsn); kid.of(Apa, ThePrsn)}
structure person.proc(                    !for person maker
  proc (pntr, string, bool,               !population of person , name, sex
  pntr, pntr, pntr) pproc)                !dad,mum,dob
s.enter("MakePerson" ++ mppw, db, person.proc(MkPerson))
...                                       !make the rest of the operations
commit()
```

for a succession of operations.

Figure 7-4: A different view (abstract data type) is established

```
structure person.proc( proc(pntr, string, bool, pntr, pntr, pntr)pproc)
structure population(pntr TbyN, TbyNme, TbySnd);
structure date(int day, month, year)
...
let db = open.database("people", "Lunga", "write");
if db is error.record do {grumble(); abort}
...
let new.person = s.lookup("Make.a.Person" ++ mppw, db)(MkPerson)
let lookupName = s.lookup("LookUpName", db)(lun)
...
let curTbyN := nil; let curTbyNme := nil; let curTbySnd := nil
let popul := nil
let GetPerson = proc(string nom → pntr)
  {let p := nil !for unknown people
   if nom ≠ "" do
      begin
      ...                                 !ask for DOB, consult soundex etc
      end else ...   ; p}
let set.pop(string popName)
  {let population := s.lookup(popName, db); if population = nil do ...
   popul := population; curTbyN := popul(TbyN); curTbyNme := popul(TbyNme);
   curTbySnd := popul(TbySnd)}
let record.birth = proc(string mumN, dadN, KidN; bool KidSex; pntr KidDoB)
  {let Mum := GetPerson(mumN); let Dad := GetPerson(dadN)
   MkPerson(popul, KidN, KidSex, Mum, Dad, KidDoB)}
structure person(int pn; string name; bool sex;
   pntr dob,mother,father; *pntr kids,spouses)
let app.to.pop( proc(string Name; pntr p)fn)
 { let lp = proc(int PN; pntr T, E → bool)
     {let Her = i.lookup(PN, T); fn(Her(name), Her); true}
   let c = i.scan(curTbyN, nil, lp)}
structure ppstr(proc(string, string, string, bool, pntr)rb;
proc(string)sp, proc( proc(string, pntr))atp; ... )
s.enter("pplePak", db, ppstr(record.birth, set.pop, app.to.pop, ...)); commit()
```

7.1.3. Performing an update

Given the view constructed above, the program to record the birth specified in our standard test case (test 4 of figure 3-5) is shown in figure 7-5. Note that in this program the structures used in the database are not made visible - the only structure being explicitly used is that which gives the required abstract type as a set of procedures.[3]

[3]The structure for dates could also have been hidden as an abstract type, so that this programmer did not know whether they were held as a record or packed in an integer.

Figure 7-5: record birth of Morwenna Trefusis

```
structure ppstr(proc(string, string, string, bool, pntr)rb;
   proc(string)sp, proc( proc(string, pntr))atp, ... )
structure date(int day, month, year)
let people = open.database("people", "Io", "write")
if people is error.record do {grumble(); abort}
let female = true; let male = false
let pplePak = s.lookup("pplePak", people)
let setpop = pplePak(sp); let record.birth = pplePak(rb)

setpop("Flushing")
record.birth("Moyana Trefusis", "Janner Curnow", "Morwenna Trefusis",
   female, date(22,9,82))
commit()
```

7.1.4. Performing retrievals with PS-algol

Figure 7-6: A PS-algol program to print the females in a population

```
structure ppstr(proc(string,string,string,bool,pntr)rb, proc(string)sp;
   proc( proc(string, pntr))atp, ... )
let people = open.database("people", "Jura", "read")
if people is error.record do {grumble(); abort}
let female = true; let male = false; let pplePak = s.lookup("pplePak", people)
let setpop = pplePak(sp); let AppToPop = pplePak(atp)
...
setpop("Flushing")
let print.female = proc(string N, pntr P)
  {if p(sex)=female do {print.name(N); print.date(dob(P))}}
AppToPop(print.female)
```

Similarly, the simple retrieval of all the females can be coded in PS-algol as shown in figure 7-6. PS-algol provides a mechanism for iteration over its "bulk" objects, tables, by providing a procedure that will apply another procedure to every entry in the table (or until the applied procedure returns **false**). This is utilised here to apply a female-printing function *print.female*.

In order to print the ancestors, it has been necessary to construct, in the PS-algol program shown in figure 7-7, a representation of two sets: the ancestors found so far, and the generation currently being considered. This has been done using the *table* construct, with one table called *Ancs* and a series of tables *kids*, *Kids*, and *parents* representing successive generations under consideration. The procedure *NextGen* is defined to find all the parents of a given generation.

Figure 7-7: A PS-algol program to list the antecedents of Janner Curnow

```
structure ppstr(proc(string, string, string, bool, pntr)rb;
  proc(string)sp, proc( proc(string, pntr))atp, ... )
structure person(int pn; string name; bool sex; pntr dob, mother, father;
   *pntr kids, spouses)
let people = open.database("people", "Jura", "read")
if people is error.record do {grumble(); abort}
let female = true; let male = false; let pplePak = s.lookup("pplePak", people)
let setpop = pplePak(sp); let AppToPop = pplePak(atp); let GetPerson = pplePak(gp)
...
setpop(Flushing)
let JC = GetPerson("Janner Curnow")
let Ancs = table(); let kids := table()          !use set property of tables
i.enter(JC(pn), kids, JC); let count := 1        !initial set
let NextGen = proc(pntr Kids → pntr)
  {let parents = table()                         !Empty set. Next generation
   let collect = proc(int PN; pntr P)            !record if never seen before
     {let Him = i.lookup(PN, Ancs)
      if Him = nil do                            !scanned already?
        {i.enter(PN, Ancs, P); i.enter(PN, parents, P)}}
   let get.new.parents = proc(int PN; pntr T, E → bool)
     {let Her = i.lookup(PN, T); let Ma = Her(mother); collect(Ma(pn), Ma)
      let Dad = Her(father); collect(Dad(pn), Dad); true}
   let c = i.scan(Kids, nil, get.new.parents); parents}
let count := 0
while count ≠ 0 do { kids := NewGen(kids); count := cardinality(kids) }
let pp = proc(int PN; pntr T, E → bool)          !print each resulting ancestor
   { print.name(Him); print.date(dob(Him)); true }
write "'nThe ancestors of Janner Curnow are:"
let c = i.scan(Ancs, nil, pp); write "'n****'n"
```

7.2. Summary and analysis of PS-algol

It appears that PS-algol is the only implemented language that provides persistence uniformly at present. That is, it is the only strongly typed procedural language that adheres without exception to the principle of data type completeness, allowing data of any type to have any persistence. The preceding examples show that this results in flexibility in the structures that can be constructed and in the operations that can be associated with them. Two important questions remain to be resolved in the context of this research:

1. Can the total systems constructed using this approach perform efficiently?

2. Will the approach work with other languages based on different type systems?

Both of these questions are still open, requiring more research. Some of the results show that the performance of these systems is potentially good, in particular that there are advantages to the lazy migration of data and meta data employed. However, with some program runs, poor performance results from having to maintain two address spaces and their relationship. This problem can be overcome using new computer architectures which operate with one (persistent and very large) address space. An unresolved aspect of obtaining efficiency is the incorporation of more concurrency into this language and its implementation. The attempts to define languages with concurrent facilities should provide much that is needed here.

The need to provide persistence in the context of other type systems is both a practical need and a goal in research. The PS-algol system exploits the open-ended polymorphism that the **pntr** type in PS-algol gives. With the language ML, which has an elegant form of polymorphism, this would not be possible, since the polymorphism is restricted to the set of previously known types. However, our interfaces to the database have to operate with all types which may be introduced in the future. We need to develop a new type system, so that higher level types can be introduced and denoted succinctly. One of the problems with the flexibility in PS-algol is that, in contrast to a system with higher level types, programs can require much detail. To obtain the higher level types without losing the flexibility is therefore an obvious goal. For example, at present we can define a set of procedures in PS-algol which give all the operations of the relational data model. What we would like to do is to define a flavour of that model as a type; relational structures and databases would just be instances of that type.

It would appear that the value of data type completeness is well demonstrated by PS-algol, as is the value of making procedures first class objects in the language. There are surely a class of problems, such as CAD and Engineering, for which the data type system of PS-algol is preferable to the more restrictive, predefined higher level data models. The question remains: is it conceptually and engineeringly feasible to get the best of both worlds?

8. CONCEPTUAL PROGRAMMING LANGUAGES

A class of languages has been developed which attempt to provide "natural" ways of describing and operating on data. That is, the constructs and operations closely mirror the concepts we use when thinking, as they are evinced in natural language. These languages have two sources: the languages developed by the Artificial Intelligence research community, and those developed by the Conceptual Data Modelling research workers who base their ideas on Semantic Networks. A good survey of such languages is

given by Borgida [Borgida, A. 83].

Typical of these languages are Taxis [Borgida & Wong, H.K.T. 81, Borgida 82, Mylopoulos *et al.* 80, Mylopoulos & Wong 80] and Galileo [Albano & Orsini 83, Albano 83, Albano *et al.* 83a, Albano *et al.* 83b]. For the purpose of exposing the principles of such languages we present examples in Galileo as it is defined in the last reference[4]. Galileo takes many of its constructs from ML, which is discussed in the next section.

8.1. The family tree example in Galileo

Figure 8-1: Definition in Galileo of an Abstract Type for "dates"

```
type date ↔ <optional (year: num and month: num and day: num)>
with
   newDate (y, m, d) := if validDate (y, m, d) then
      mkDate (<(year := y and month := m and day   := d)>)
      else failwith "newdate:invalid"
and
   newUnknownDate := mkDate (<nil>)
and
   before (d1: date, d2: date) := ...check if d1 is before d2...
drop mkDate
```

To investigate the standard problems (given in figure 3-5), consider first the representation of a date in Galileo. As in PS-algol, data structures and the operations on them are introduced together. Figure 8-1 shows this for dates which have values of two types: unknown values and those constructed from three numbers. **Drop**ping the standard constructor for dates (*mkDate*) prevents programmers from constructing invalid dates. This leaves the two functions *newDate* and *newUnknownDate*, declared in the scope of *date*, as the only available date constructors.[5]

8.1.1. A data structure for family trees in Galileo

Within the scope of figure 8-1, the declarations of figure 8-2 would construct an abstract data type for family trees. Here a **class** *Persons* together with a representation

[4] which has been submitted for publication in the ACM Transactions on DataBase Systems.

[5] **type** introduces a type declaration, **and** introduces another declaration at the same level whereas **with** starts the next declaration as the first declaration in an inner scope. Note that the record structure of the *known* option of *date* is a scope holding three names paired with three integer values. The construct **failwith** raises an exception, with the string that follows as its value.

Figure 8-2: A Galileo definition of an abstract type for family trees

```
FamilyTree := (rec Persons class
 (Person ↔ (name: string and dOfB: var date
   and dOfD: var date and parents: var seq Person
   ext father := derived
     first(all p in at parents of this with p alsoIn Males)
       if fails failwith "father:unknown"
   and mother := derived
     first(all p in at parents of this with p alsoIn Females)
       if fails failwith "mother:unknown"
   and kids := derived all p in Persons with this into at parents of p)
 with
   newPerson(n: string, db, dd: date, par: seq Person) :=
     if length (par) > 2 then failwith "newPerson:tooManyParents"
     else if length (all p in par with p alsoIn Males) > 1 then
       failwith "newPerson:tooManyFathers"
     else if length (all p in par with p alsoIn Females) > 1 then
       failwith "newPerson:tooManyMothers"
     else if length (all p in par with before(db,dOfB of p)) > 1 then
       failwith "newPerson:bornBeforeParent"
     else if before (dd, db) then failwith "newPerson:deathBeforeBirth"
                                          ... other validity checks ...
       else mkPerson(n, var db, var dd, var par)
                                                ... other operators ...
   drop mkPerson)
 key(Name)
 and Males partition of Persons with Females class Male ↔ is Person
 and Females partition of Persons with Males class Female ↔ is Person)
```

for individuals of that class are introduced[6] . The explicit separation of the name for the extent of a type (*Persons*, *Males* and *Females* in this case) from the name for the type (*Person*, *Male* and *Female*), is a considerable improvement over DAPLEX.[7] The series of tests shown check the validity and compatibility of the data associated with a person, raising a named exception if any invalidity is detected. This is similar to the abstract type protection in PS-algol. An alternative mechanism, specified by **assert** <*predicate*>, associates checks with the abstract type. The advantage is that they are then not explicitly included in the definition of each operation, but they then may be more difficult to understand when the checks fail, making it more difficult to program recovery. It is, however, clearer what checks apply, and there is no risk that they may

[6] **rec** (short for recursive) allows the name that is about to be declared to be used in its own definition.

[7] **var** designates a reference component, so that its value may be replaced by assignment, other values are held constant. PS-algol has the same features with a different notation. **at** performs the dereference operation obtaining the contents of such a variable. The bulk data objects are sequences, thinly disguised ML lists. The **all** construct shown constructs a new sequence of the elements satisfying the predicate from the given sequence (classes are defined to be sequences). The **alsoIn** operator is the set membership test.

be omitted in certain operations. The two classes *Males* and *Females* are constructed as a **partition** of *Persons*, with the property that all members of these classes are also members of *Persons* and the property that the intersection is empty. Unlike ADAPLEX, but like DAPLEX, a member of *Persons* does not have be a member of either subclass. The types *Male* and *Female* are subtypes of *Person* inheriting that type's properties; they could also have new properties.

Unlike the treatment in PS-algol, the parents have been represented as a sequence, and the kids are not explicitly recorded. The functions to obtain *father*, *mother* and *kids* then **extend** the scope, introducing derived fields reminiscent of DAPLEX, but not requiring any special linguistic concept.

8.1.2. Retrieving data in Galileo

In the context of the definitions given in figure 8-2, the program to print the list of females is trivial, as is shown by figure 8-3. Like ML, Galileo is an interactive language, and the value of the last expression will be printed. In this case, the expression is a sequence constructed from the subclass *Females*.

Figure 8-3: A Galileo program to print the names of the females

```
for p in Females loop name of p
```

This is as concise[8] and understandable as conventional query languages, but the full power of the type system and computational model of ML is also available.

Figure 8-4: A Galileo program to retrieve the ancestors of Janner Curnow

```
ancs( p: Person) :=
    use rec trans (y, visiteds) :=
      if y into visiteds then visiteds
        else collect (at parents of y, [y] append visiteds)
    and collect(l,visiteds) :=
      if emptyseq l then visiteds
        else collect (rest (l), trans (first (l), visiteds))
    in
      trans(p, []);
ancs (get Person with name = "Janner Curnow")
```

A simple recursive function using the sequences as sets (**into** facilitates this by providing a test of set membership) can be constructed to compute any transitive closure over any lattice. A slightly more specific function *trans* is used in figure 8-4 to collect all the ancestors. *trans* terminates recursion if the person has been seen before (is in

[8] In fact Galileo allows the contracted form: **for** *Females* **loop** *name*

visiteds) and otherwise uses *collect* to apply itself to all the parents (if any) found, having inserted them into *visiteds* first. The **get** construct guarentees that exactly one valid instance is selected from the (implicitly) specified extent. The code to actually print the set of ancestors has been omitted, so some default printing of the sequence would therefore take place.

8.1.3. Update in Galileo

Figure 8-5: A Galileo program to record a birth

```
use p := get Person with name = "Janner Curnow" if fails failwith "PaUnknown"
and m := get Person with name = "Moyana Trefusis" if fails failwith "MaUnknown"
ext Morwenna := newPerson("Morwenna Trefusis", newDate (1983, 11, 1),
                          newUnknownDate, [p; m])
in
  inFemales (Morwenna, Morwenna)
```

The snippet of program, again assumed to be in the scope of earlier declarations, shown in figure 8-5 demonstrates how the standard update might be coded. We already have an update operation defined which checks constraints and which cannot be bypassed. Locating the parents is here assumed to be simple, using the **get** statement already introduced.

8.2. Analysis of the Conceptual Programmming languages

Prior to the definition of Galileo, it was clear that the languages in this class were difficult to formulate. Those associated with SDM [Hammer & McLeod 81] and Taxis introduced many good ideas, each individually useful and reasonable to define intuitively. However, as soon as the interaction between the various ideas were considered, they became difficult to understand and even harder to formulate precisely. This would have inhibited useful implementations, and probably have led to many surprises when such languages were used. Galileo has made good progress in overcoming this problem without abandoning the conceptual modelling constructs. This has been achieved by building on two standard programming language concepts: polymorphic types and environments. Both have been precisely formalised and powerfully reallised in ML, and Galileo inherits and carefully extends their form. A certain comfort can be gained from the precise specification of these ideas and hence of the full semantics of Galileo. The design work continues at the University of Pisa, and the implementation is being undertaken by various Italian companies.

There is no evidence, in the examples, nor in the preceding discussion, of the arrangements for providing persistence. As the language treats persistence as an orthoganal property of data, it is not necessary to make this obtrusive. Again the

advantage of the principle of data type completeness is apparent. The definitions which reside in a standard global scope are those which persist. As they will normally be the definition of a hierachy of abstract type and module definitions, the name space and the data space are quite general "databases" of instances of Galileo's types. The extensive set of environment operators[9] allow a sophisticated space of bindings to be constructed and used. It is not clear that the necessary rebinding arrangements are there to permit repair of program when data already populates the database. The design of atomic transactions, authorisation, and concurrency is not yet complete. It is not clear that independently constructed bodies of data can ever be used together, though this is often a requirement. But certainly the language allows concise and precise specification of data structure and algorithm in a natural way and these are properly bound together.

---

[9] The environment operators of Galileo are:

1. Id := term *introduces a binding between a name* id *and a type or value.*
2. A **and** B *introduces the bindings of both* A *and* B *but each may not use the other.*
3. A **ext** B *define* B *using the environment* A *then introduce the bindings of* B *and those of* A *that have not been redefined.*
4. **rec** A *allow the bindings of* A *to be used in the definition of* A *then introduce* A.
5. **type** A *introduce only the bindings in* A *which bind names to types.*
6. A **drop** Id *introduce the bindings of* A *without the binding introduced by* Id.
7. A **take** Id *introduces only the binding associating* Id *from* A.
8. A **rename** Id **by** NewId *introduces the bindings of* A *except that the one concerning* Id *is renamed.*
9. **type** Id <=> Type *introduces a new type* Id *isomorphic with* Type *and two transfer functions* mkId *and* repId *into the environment*
10. **type** Id <-> Type *as for* <=> *except that the operators on* Type *are available on* Id.
11. **use** Binding *extends the global environment with* Binding
12. **enter** Id *arrange that subsequent evaluation takes place within the environment bound to* Id.
13. **quit** *move back to the global environment*
14. A **with** B *constructs the bindings in* B *in the context of* A *and exports the bindings of* B *and the type bindings only of* A.

This should be contrasted with a typical algol like language which has the equivalent of just two environment operators. One does wonder whether quite so much variety is necessary.

9. APPLICATIVE PROGRAMMING DEVELOPMENTS

There are many candidate languages that do not explicitly aim to develop persistent programming, but which are contributing ideas that may be influential to the development of persistent programming. In the last section, the influence of ML on Galileo was evident. KRC [Turner, D.A. 81] has influenced Buneman's development of FQL [Buneman & Davidson 83, Nikhil 82, Nikhil 84]. PROLOG, as it uses the same construct for data and rules, has some attractive properties when persistent programming is considered.

9.1. Developing ML

The neat and well defined type system of ML [Gordon *et al.* 79] was an important contribution to Galileo. Recently there have been significant improvements in the language's notation: introducing patterns from Hope [Burstall *et al.* 80]. This began with a much improved implementation of ML on the VAX [Cardelli 83a]. It has continued with a redesign of the language to define standard ML [Cardelli 83b, Milner 83].

9.1.1. Examples in standard ML

The first two examples are intended to illustrate the power of polymorphism. They allow the construction of sets of anything and lattices of anything[10] . Note that when a lattice is parameterised, the sets used to construct it are automatically parameterised with the same type. Figure 9-1 sets up an abstract type for sets represented as lists. Figure 9-2 then defines an abstract type for a lattice and a transitive closure operation using these sets over it. By suitably parameterising these abstract types, they may be used to represent our family tree. Note that infix operators were defined for sets, so that operations of the form shown in figure 9-3 are then possible. The abstract type hides the implementation so that later it may be replaced by a more efficient one. Both applicative and procedural style code are illustrated in figure 9-1. patterns are used to decompose structures and the functions are defined for specific cases and then for the remaining more general case.

Given these definitions, with people being represented by a tuple of type **string*bool*date*date** for name, sex, date of birth and date of death respectively, the set of females may be printed by the code in figure 9-4 and the set of ancestors by the code in figure 9-5.

[10] Actually not quite anything, as a set of instances of an abstract type cannot be constructed as equality is used in the set definitions and is not defined automatically for abstract types. An abstract type parameterised with the equality function could be written, so that these sets could be constructed, but then the simple syntax of infix operators would not be achieved.

Figure 9-1: An abstract ML type for sets

```
infix contains; infix member; infix union; infix insertedinto;
infix without; infix subset; infix superset; infix suchthat;
abstype α set = set of α list
with val EmptySet = set nil
   and rec mkSet x = set [x]            {given an element make a set of it}
   and empty (set x) = null x           {test for the empty set}
   and choice (set []) = escape "can't choose from the empty set"
     | choice (set(x :: X)) = x, (set X)
   and x member (set nil) = false
     | x member (set (y :: Y)) = (x = y) or (x member (set Y))
   and (set nil) contains (set nil) = true
     | (set (x :: _)) contains (set nil) = true
     | (set nil) contains (set (x :: _)) = false
     | S contains (set (y :: Y)) = (y member S) & (S contains (set Y))
   and X subset Y = Y contains X
   and X superset Y = X contains Y
   and x insertedinto (set Y) = set (if x member (set Y) then Y else x :: Y)
   and ListToSet [] = set []            {construct a set from a list of elements}
     | ListToSet (x :: X) = x insertedinto ListToSet X
   and (set nil) union Y = Y
     | (set (x :: X)) union Y = x insertedinto (set X union Y)
   and (set nil) suchthat pred = set nil
     | (set (x :: X)) suchthat pred = if pred(x) then
                                        x insertedinto ((set X) suchthat pred)
                                      else (set X) suchthat pred
   and X without (set nil) = X          {set difference}
     | (set X) without Y =              {demonstrate nonapplicative ML}
       let val res = ref (set []) and todo = ref X
       in
          (while not (null (!todo)) do
             let val first :: rest = (!todo)
             in
               (if not (first member Y) then res := first insertedinto (!res)
                                        else res := !res;
                todo := rest) end;
           !res) end
   and MapSet ((set nil), f) = set nil
     | MapSet ((set (x :: X)), f) = (f x) insertedinto (MapSet ((set X), f))
   and AppToSet ((set nil), f) = ()
     | AppToSet ((set (x :: X)), f) = (f x; AppToSet ((set X), f))
end;
```

9.2. Analysis of Applicative languages

Space has precluded the examination of many interesting languages. ML, which, as was demonstrated, is not always applicative, represents an example of a good modern language. The abstract types and modules (not illustrated) provide a good deal of extensibility. New higher level types such as set and lattice can now be defined quite

Figure 9-2: An abstract data type for any lattice

```
abstype rec α lattice = lattice of (α * ((α lattice) set))
with val lattice NodeInfo (Id, _) = Id
 and lattice NodeAncestors (_, Ancs) = Ancs
 and transitive StartNode =
 let val rec trans ToVisit HaveFound =
    if empty ToVisit then HaveFound
       else
          let val (lattice (ThisOne, NextLot)), rest = choice ToVisit
          in
             trans (rest union NextLot) (ThisOne insertedinto HaveFound) end
 in
    trans (mkSet StartNode) EmptySet end end;
```

Figure 9-3: Examples of using the abstract type for sets in ML

```
val A = mkSet "a";                    {a set of strings, 1 element "a"}
val AB = "b" insertedinto A;          {construct a new set of two strings}
val C  = mkSet "c";                   {a set of one string "c"}
val ABC = AB union C;                 {set of 3 strings "a", "b", "c"}
"a" member C, "a" member AB, AB superset A, AB superset ABC;
    {The preceding expression returns false, true, true, false}
val X = ListToSet [1; 2; 3; 5; 7; 11]     {a set of integers}
val HailStone = ListToSet [1,0,1; 2,1,2; 3,7,16; 7,16,52]
                                      {A set of integer triplets}
```

Figure 9-4: ML program to print the list of females

```
val female (_, sex, _, _) = sex;
val PrintName (name, _, _, _) = (output(terminal, "\L"); output(terminal, name));
AppToSet ((People suchthat female), PrintName);
```

Figure 9-5: Printing a list of the ancestors in ML

```
val JC, _ = choice (people suchthat fun (name, _, _, _).name = "Janner Curnow");
AppToSet  ((transitive JC), PrintName);
```

succinctly. But this power of definition is limited. The definitions above will only work for sets of objects of concrete type. So if we defined person as an abstract type, which we naturally would do, then we could not construct a set of persons with the above definition. The reason for this is that equality, used in the *member* definition (see figure 9-1), is not defined for abstract types. Because of this, future versions of the ML compiler may disallow such a definition. To overcome this one has to parameterise all the set operations with *eq*: $\alpha * \alpha \rightarrow$ **bool** to be used for equality tests. This becomes more verbose and loses the infix notation. It is perhaps just the tip of the iceberg. There are many higher order types, such as relations, which one can't readily define succinctly in ML, despite its polymorphism. One might define a relation as a set of tuples, but the general type relation allows any tuple type and size. When relational operations, such as join are applied, there are type rules to be enforced, not describable

in ML. Achieving the ability to define such types and their operators remains an active area of research.

ML has contributed a powerful model of polymorphism and an extremely clever compilation mechanism to ensure that all type checking is static. It is possible that to get all the flexibility we need for higher level types we will have to allow some checking to be postponed. Incremental compilation is used to provide efficient interactive operation. Thus if ML had persistence, the query language roles from trivial to sophisticated queries would already be met by the language itself.

ML has made explicit a number of environment manipulations, and the programmer is aware of the bindings he creates and when and where they are available. As we progress to persistent languages, it seems certain that we will want such a conciousness of the environment in which code is written. We will also want operators which extend ML's set of environment operations to allow environment modification. Otherwise, populated databases cannot be modified to meet revised needs.

A particular strength of ML is the very small number of primitive concepts which underlie the language: a small number of base types, functions and tuples, environment manipulation and patterns. All other aspects of the language are explained in terms of these. It is a sign of progress that the language has become so much more simple, and yet powerful and precisely defined. It is necessary that such a pattern influences persistent language design.

KRC [Turner, D.A. 81] introduced new programming power, in permitting succinct expressions defining computation on infinite sets (or more precisely sequences). Can this be combined with the strengths of ML? Prolog allows the implicit definition of search strategies (its other properties, based on patterns are largely in ML) and allows these to be used in any direction and generality, with the possibility of backtracking. Would this be useful or confusing with very large volumes of data? Certainly queries can be expressed very succinctly, but the implications of an update and the combinatorial effects in a search may be difficult to understand. This may be the reason why the Prolog programmer appears to have to know how Prolog is evaluated [Colmerauer 83].

The search for more powerful constructs [Cardelli 84] and more powerful computation mechanisms [Ait-Kaci 83] continues, and will hopefully always inform language design.

10. CONCLUSION

The sequence of languages presented give clear evidence of progress. In analysing each language the reader has been asked to step outside his normal use of a programming language - to look at parts of a language without concern for the whole, to look at the language without being put off by the unfamiliarity of its notation, to try to suppress the effect that the languages that are like those one has already used, appear more natural than those that introduce radical concepts, and to try to consider general usage while looking at examples based on only one simplified task. This is difficult, and as authors we are not immune to these problems. But the value of previous understanding and experience is not negligible. Consequently we should preserve this value if there are not greater gains to be made from accepting change.

The initial "languages" were standard programming languages with collections of routines that could be called. These collections of routines became progressively more formalised and consistent, and possibly more powerful. To use them they were embedded in the host language with a notation that suited them, but which was unrelated to the notation of the host language. This reduced the labour of using them, but did not reduce the conceptual difficulties arising from different name spaces, different notations, and different type regimes. It did not permit any check that the programmer was behaving consistently *as he wrote a program*. The program could not be *understood* from just the program text.

The integrated languages made inroads into this problem, but only when all the data structures that crossed the database to program interface were those to which the implementor had given a full range of persistence. The problem remained of the programmer having to explicitly manage the relationship between two representations for all other data. But in these languages notation was consistent, type checking rules could be applied and programs carried with them enough information that they could be understood without recourse to other texts.

The persistent programming languages have made that treatment consistent for all data of all types. This means that the programmer is not constrained by the preconceptions of the implementor about what data is useful in a database. The management of representation translation is no longer necessary. An important step here was to make the function a data object which then had the usual rights to persistence. This allows program and data to remain bound together and for sophisticated data hiding and mappings to be implemented. It would be a significant component of a general and flexible protection mechanism. These languages do not yet

have a succinct notation for higher level data types, nor do they yet provide well for concurrency.

In the search for better data types we look at the conceptual programming languages. The progress with this research has led to a number of language proposals and partial implementations. There is a consensus that classes of objects and type hierarchies are useful. But there is not yet a consensus over the choice between sets and sequences or whether both need be provided. The language Galileo was considered a good representative, as it has the properties of a good programming language with abstract types and modules. It allows all data to be persistent and uses the module as the unit of persistence. It, too, takes advantage of the power of persistent functions. It is substantially based on the mostly applicative language ML. This is a demonstration that the data base community is being influenced by the programming language community. This is definitely progress. The inverse influence is also needed. Ultimately these two research communities should merge. All large scale systems, and hence most programming, involves persistent data. Consequently, programming language design should meet the need for persistent data. All databases need programs to be written and run against them. Hence, the database community needs good language design. Ultimately, all database concepts will be embedded within the semantics and notation of the programming language and will rarely be *consciously* used. That, perhaps, is the goal of persistent programming research. The analogy with the management of heap space is obvious. In more primitive languages, the programmer is responsible for heap management, and uses it explicitly. In a modern langauge like ML or PS-algol, one is rarely aware when one is constructing an object on the heap.

A glance at the welter of current programming language research (much of it is in applicative and logic languages at present but could be elsewhere) shows that there are still unsolved problems to challenge and new ideas to assimilate. The language ML shows that the idea of environments and environment manipulation is important. But does it go far enough? Should environments be nameable first class objects in a persistent language? This would meet some of the needs for identifying and changing persistent definitions. ML showed how a language can be built from a parsimonious use of concepts to be very general and powerful. But it does not yet allow us to define our own high level types that are popular for databases. Can the ideas of polymorphism be extended to this level? If not, then we should find some other way of reaching a state where we can just declare the data types that correspond to our data models, possibly by accepting some run time type/consistency checking. Until we do that, we will find our persistent languages both over-complex and restrictive, and our research into types and

data models will be speculative rather than based on sound experimental technique, as we will not be able to afford the implementations necessary for those experiments.

ML is interactive, and incrementally checked and compiled. It is certain that future persistent languages must have this property, so that query languages cease to be a special case.

The importance of consistency has been frequently observed. The general applicability of rules in both PS-algol and ML give these languages considerable power, and make them easy to understand, learn and use. Of particular importance to the design of persistent programming languages is the orthogonal availability of persistence. It should be accepted that, in all persistent programming languages, the principle of data type completeness will be adhered to, so that data of any type may be persistent or transient. If this proves hard to achieve, either to obtain performance or because it conflicts with our preconceptions about *first normal form relations*, then it should stimulate research. The principle should not be abandoned. PS-algol has demonstrated it can be implemented and Galileo has demonstrated that it can be formally described.

11. Acknowledgements

The work was supported by a number of grants from the British Science and Engineering Research Council and from ICL.

The presentation and content has benefitted from many discussions with colleagues and students in Edinburgh and Philadelphia. The underlying themes have been refined in discussion with Peter Buneman, who has made many helpful suggestions particularly the Adaplex examples. Steve Fox, of CCA, kindly read and redesigned those examples. Rishiyur Nikhil contributed initial examples for Galileo and helped form those for ML. He, Larry Krablin and Luca Cardeli also read a draft and made a large number of suggestions for improvement. Discussions with Mark Reinhold and Sharon Perl contributed to the language examples. Renzo Orsini and Alex Borgida suggested improvements to the section on conceptual languages. Peter Gray assisted with the Fortran/ CODASYL examples.

References

[Ait-Kaci 83] Ait-Kaci, H.
Outline of a calculus of type subsumptions.
Technical Report CIS-83-34, Dept of Computer and Information Science, The Moore School of Electrical Engineering, University of Pennsylvania, December, 1983.

[Albano 82] Albano, A., Occhiuto, M.E. and Orsini, R.
A uniform management of persistent and complex data in programming languages.
In Atkinson, M.P. (editor), *Pergammon Infotech State of the Art Report, Series 9, Number 4: Database*, pages 321-344. Pergammon Infotech, Maidenhead, England, 1982.

[Albano 83] Albano, A.
Type Hierachies and Semantic Data Models.
ACM SIGPLAN Notices 18(6):178-186, June, 1983.
Presented at the Symposium on Programming Language Issues in Software Systems, San Francisco, California June 27-29 1983.

[Albano & Orsini 83] Albano, I. and Orsini, R.
Dialogo: An interactive environment for Conceptual Design in Galileo.
In Ceri, S. (editor), *Methodology and Tools for Database Design*, pages 229-253. North Holland, Amsterdam, 1983.

[Albano *et al.* 83a] Albano A., Capaccioli, M., Occhiuto, M.E. and Orsini, R.
A modularisation mechanism for conceptual modelling.
In *Proceedings of the ninth international conference on Very Large Data Bases.* October, 1983.

[Albano *et al.* 83b] Albano, A., Cardelli, L. and Orsini, R.
Galileo: A Strongly Typed Interactive Conceptual Language.
Technical Report 83-11271-2, Bell Laboratories, Murray Hill, New Jersey, July, 1983.
Submitted to ACM transactions on Database Systems.

[Atkinson 74] Atkinson, M.P.
Survey of current research topics in Data-structures for CAD.
National Computer Centre, Manchester, England, 1974, pages 203-217 and 297-303.

[Atkinson 78] Atkinson, M. P.
Programming languages and databases.
In S. B. Yao (editor), *The fourth international conference on Very Large Data Bases,Berlin, West Germany*, pages 408-419. September, 1978.

[Atkinson *et al.* 81]
Atkinson,M.P.,Chisholm,K.J. and Cockshott,W.P.
PS-algol: an Algol with a Persistent Heap.
ACM SIGPLAN Notices 17(7), July, 1981.
Also available as Tecnical Report CSR-94-81, Computer Science Department, University of Edinburgh.

[Atkinson *et al.* 83a]
Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. and Marshall, R.M.
Algorithms for a Persistent Heap.
Software Practice and Experience 13(7), March, 1983.

[Atkinson *et al.* 83b]
Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R.
An Approach to Persistent Programming.
Computer Journal 26(4), November, 1983.

[Atkinson *et al.* 83c]
Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P.
CMS-A Chunk Management System.
Software-Practice and Experience 13:273-285, 1983.
Also available as Technical Report CSR-110-82, Computer Science Department, University of Edinburgh.

[Berry 81] Berry, D.M.
Remarks on R.D. Tennant's **Language Design Methods Based on Semantic Principles**: Algol-68, a Language Designed using Semantic Principles.
Acta Informatica 15:83-98, 1981.

[Borgida 82] Borgida, A.
Features of Conceptual Models for Information Systems - a survey.
In *Proceedings of the 1st AUC Conference, Medellin, COLOMBIA.* September, 1982.
Also available as a Technical Note from Department of Computer Science, Rutgers University.

[Borgida & Wong, H.K.T. 81]
Borgida, A. and Wong, H.K.T.
Data models and data manipulation languages: complementary semantics and proof theory.
In *The seventh international conference on Very Large Data Bases, Cannes, France*, pages 260-271. September, 1981.

[Borgida, A. 83] Borgida, A.
Features of Languages for Conceptual Information System Development.
Technical Report, Department of Computer Science, Rutgers University, Hill Center, Rutgers University, New Brunswick, New Jersey 08903, USA, 1983.

[Brinch-Hansen 75]
Brinch-Hansen, P.
The programming language Concurrent Pascal.
IEEE transactions on software engineering :199-207, June, 1975.
Also appears in Horowitz 1983 pages 264-272.

[Brooker 82] Brooker, R.A.
A 'Database' Subsystem for BCPL.
The Computer Journal 25(4):448-464, 1982.

[Buneman & Davidson 83]
Buneman, P. and Davidson, S.
Database Research at the University of Pennsylvania.
In *Proceedings of the Digital Equipment Corporation workshop on Databases: Boston.* April, 1983.

[Buneman *et al.* 82]
Buneman, P., Herschberg, J. and Root, D.
A CODASYL interface for Pascal and Ada.
In *Proceedings of the second British National conference on Databases: Bristol, England.* July, 1982.

[Burstall *et al.* 80]
Burstall, R.M., MacQueen, D.B., and Sannella, D.T.
HOPE: An Experimental Applicative Language.
Technical Report CSR-62-80, University Of Edinburgh, 1980.

[Cardelli 83a] Cardelli, L.
The Functional Abstract Machine.
Technical Report, Bell Laboratories, Murray Hill, New Jersey 07974, March, 1983.

[Cardelli 83b] Cardelli, L.
ML under UNIX: pose 3.
Technical Report, Bell Laboratories, Murray Hill, New Jersey 07974, November, 1983.

[Cardelli 84] Cardelli, L.
A semantics of Multiple Type Inheritance.
private communication , 1984.

[Childs 68] Childs, D.L.
Description of Set Theoretic data structure.
proceedings of the AFIPS FJCC 33, part 1:557-564, december, 1968.

[Cockshott *et al.* 83]
Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morrison, R.
The persistent object management system.
Software Practice and Experience 14, 1983.
Also as Technical Report PPR-1-83, Computer Science Department, University of Edinburgh.

[CODASYL 71] Codasyl Committee on Data System Languages.
CODASYL Data Base Task Group Report.
Technical Report, ACM, 1971.

[Codd 70] Codd, E.F.
A Relational Model for Large Shared Databanks.
Communications ACM 13(6):377-387, 1970.

[Codd 79] Codd,E.F.
Extending the Relational Model of Data to Capture More Meaning.
ACM Transactions on Database Systems 4(4), December, 1979.

[Cole & Morrison 82]
Cole, A.J. and Morrison, R.
An Introduction To Programming With S-algol.
Cambridge University Press, 1982.

[Colmerauer 83] Colmerauer, A.
Prolog in 10 figures.
In *Proceedings of the eigth international joint conference on artificial intelligence, Karlsruhe, West Germany,* pages 487-499. August, 1983.

[Computer Corporation of America 83]
Smith, J.M., Fox, S and Landers, T.
ADAPLEX: Rationale and Reference Manual
second edition, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts 02142, 1983.

[Cormack 83] Cormack, G.V.
Extensions to Static Scoping.
ACM SIGPLAN Notices 18(6):187-191, June, 83.
presented at The SIGPLAN '83 symposium on programming language issues in software systems, San Francisco, California.

[Darlington *et al.* 82]
Darlington, J., Henderson, P., and Turner, D.A.
Functional programming and its applications.
Cambridge University Press, Cambridge, England, 1982.

[Date 81] Date, C.J.
Referential Integrity.
In *The seventh international conference on Very Large Data Bases, Cannes, France.* VLDB, 1981.

[Dodd 69] Dodd, G. D.
Elements of data management systems.
Association of Computing Machinery Computing Surveys 1:117-133, June, 1969.

[Falkoff & Iverson 73]
Falkoff, A.D. and Iverson, K.E.
The design of APL.
IBM Journal of Research and Development 10?:324-334, July, 1973.

[Feldman 76] Feldman, S.I.
A FORTRANer's lament: Comments on the draft proposed ANS Fortran Standard.
ACMSIGPLAN 11:25-34, December, 1976.

[Gordon *et al.* 79]
Gordon, M. J., Milner, A. J. R. G., and Wadsworth, C. P.
Lecture Notes in Computer Science. Volume 78: *Edinburgh LCF.*
Springer-Verlag, 1979.

[Gray 67] Gray, J.C.
Compound data structures for computer aided design: a survey.
In *Proceedings 22nd Anniversary Conference of the ACM,* pages 355-365. Association of Computing Machinery, 1967.

[Gray 81] Gray, P.M.D.
Use of automatic programming and simulation to facilitate operations on CODASYL databases.
In M. P. Atkinson (editor), *Database,* pages 345-369. Pergamon Infotech, 1981.

[Gray 82] Gray, P.M.D.
The Group-By operation in Relational Algebra.
Technical Report, University of Aberdeen. Dept. of Computer Science, 1982.

[Gregains & Wiseman 84]
Gregains, A. and Wiseman, N.E.
A persistent heap for BCPL.
Software Practice and Experience to be published, 1984.

[Hall 83] Hall, P.A.V.
A persistent version of ADA.
ADA letters ?, 1983.

[Hammer & McLeod 81]
Hammer, M. and McLeod, D.
Database Description with SDM: A Semantic Database Model.
ACM transactions on Database Systems 6(3), Sept, 1981.

[Henderson 80] Henderson, P.
Functional Programming Application and Implementation.
Prentice Hall, New Jersey, Englewood Cliffs, 1980.

[Hoare 73] Hoare, C.A.R.
Hints on Programming Language Design.
Technical Report CS-73-403, Stanford University, Computer Science Department, December, 1973.

[Hoare 78] Hoare, C.A.R.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.
also in Horowitz 1983 pages 306-318.

[Horowitz & Kemper 83]
Horowitz, E. and Kemper, A.
AdaPel: A relational Extension of Ada.
Technical Report TR-83-218, University of Southern California, Department of Computing Science, Los Angeles, California, USA, November, 1983.

[Ichbiah *et al.* 79]
Ichbiah *et al.*.
Rationale of the Design of the Programming Language Ada.
ACM Sigplan Notices 14(6), 1979.

[Jarke & Koch 82]
Jarke,M. and Koch,J.
A Survey of Query Optimization in Centralized Database Systems.
Technical Report, Center for Research on Information Systems, New York University, November, 1982.
CRIS 44, GBA 82-73 (CR).

[Jarke & Koch 83]
Jarke, M. and Koch, J.
Range nesting: a fast method to evaluate quantified queries.
In *Proceedings of the ACM SIGMOD international conference on Management of Data, San Jose, California.* Association of Computing Machinery, May, 1983.
Also as a technical report from: the Center for Research on Information Systems, New York University CRIS 49, GBA 83-25 (CR).

[Jensen & Wirth 75]
Jensen, K. and Wirth, N.
PASCAL user manual and report.
Springer-Verlag, Berlin, 1975.

[Jones & Liskov 78]
Jones, A.K. & Liskov, B.H.
A language extension for expressing constraints on Data Access.
Communications of the ACM 21(5):358-367, May, 1978.

[Kent 78] Kent, W.
Data and Reality.
North-Holland, 1978.

[King 82] King, T.J.
The use of a relational database management system to store historical records.
In M.P. Atkinson (editor), *State of the Art Report Database series 9 Number 8,* pages 425-434. Pergammon-Infotech, Maidenhead, England, 1982.

[Lampson 83] Lampson, B.W.
A λ calculus of types.
Private communication , 1983.

[Lang & Gray 68]
Lang, C.A. & Gray, J.C.
ASP - a ring implemented associative structure package.
Communications of the ACM 11(8):550-555, August, 1968.

[Liskov, B. *et al.* 83]
Liskov,B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R. and Weihl, W.
Preliminary ARGUS reference manual.
Technical Report Memo 39, Programming Methodology Group, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139, USA, October, 1983.

[Mall *et al.* 83] Mall, M., Schmidt, J.W. and Reimer, M.
Data Selection, Sharing, and Access Control in a Relational Scenario.
In Brodie, M.L., Mylopoulos, J.L. and Schmidt, J.W. (editors), *Perspectives on Conceptual Modelling*, . Springer-Verlag, Berlin Heidelberg New York, 1983.
Presented at the Symposium on Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages, Intervale, New Hampshire, June 1982.

[Milne & Milner 79] Milne, G. and Milner, R.
Concurrent processes and their syntax.
Journal of the ACM 26(2):302-321, April, 1979.

[Milner 79] Milner, R.
Flowgraphs and flow algebras.
Journal of the ACM 26(4):794-818, October, 1979.

[Milner 83] Milner, R.
A proposal for standard ML.
Technical Report, University of Edinburgh, Department of Computer Science, James Clerk Maxwell Building, The King's buildings, Edinburgh EH9 3HD, Scotland, November, 1983.

[Mylopoulos & Wong 80] Mylopoulos, J. and Wong, H.K.T.
Some features of the Taxis data model.
In *The sixth international conference on Very Large Data Bases, Montreal, Canada.* November, 1980.

[Mylopoulos *et al.* 80] Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T.
A Language Facility for Designing Database Intensive Applications.
ACM Transactions on Database Systems 5(2), June, 1980.

[Nijssen 80] Nijssen, G.M.
Database Semantics.
In Atkinson,M. (editor), *Infotech State of the Art Report on Database*, . Infotech, 1980.

[Nikhil 82] Nikhil, R.
RDB - A Relational Database Management System.
Technical Report, Department of Computer Science and Information Science, University oh Pennsylvania, January, 1982.
User Manual.

[Nikhil 84] Nikhil, R.
An Incremental, Strongly Typed Applicative Programming System for Databases.
PhD thesis, University of Pennsylvania, Department of Computing and Information Science, 1984.

[Olle 78] Olle, T.W.
The CODASYL approach to Data Base Management.
Wiley Intersience, New York, 1978.

[Organick 72] Organick, E.I.
The MULTICS System.
MIT Press, Cambridge, Massachusetts, 1972.

[Parnas 72] Parnas, D.L.
A Technique for Software Module Specification With Examples.
Communications of the ACM 15(5):330-336, May, 1972.

[Richard & Ledgard 77] Richard, F. and Ledgard, H.
A Reminder for Language Designers.
ACM SIGPLAN notices 12(12):73-82, December, 1977.

[Richards & Whitby-Strevans 79] Richards, M. and Whitby-Strevans, C.
BCPL-the language and its compiler.
Cambridge University Press, Cambridge, England, 1979.

[Rowe 80] Rowe, L.A.
Reference Manual for the programming language RIGEL.
Technical Report, University of California at Berkeley, Department of Electrical Engineering, 1980.

[Rowe & Shoens 79] Rowe, L. and Shoens, K.
Data Abstraction, Views and Updates in RIGEL.
In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71-81. ACM-SIGMOD, 1979.

[Schmidt 77] Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM Transactions on Database Systems 2(3):247-281, September, 1977.

[Schmidt 78] Schmidt, J.W.
Type Concepts for Database Definition.
In Shneiderman, B. (editor), *Databases: Improving Usability and Responsiveness*, . Academic Press, 1978.

[Shipman 81] Shipman, D.W.
The Functional Data Model and the Data Language DAPLEX.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[Shopiro 79] Shopiro, J.E.
THESEUS - A Programming Language for Relational Databases.
ACM Transactions on Database Systems 4(4), December, 1979.

[Smith *et al.* 80a] Smith, J.M. *et al.*.
Multibase--Integrating Heterogeneous Distributed Data Bases.
Technical Report, Computer Corporation of America, November, 1980.

[Smith *et al.* 80b]
Smith,J.M., Bernstein,P.A., *et al.*.
Basic Architecture of Multibase.
Technical Report, Computer Corporationof America, November, 1980.

[Stonebraker *et al.* 76]
Stonebraker, M., Wong, E., Kreps, P., and Held, G.
The Design and Implementation of INGRES.
ACM transactions on Database Systems 1(3):189-222, September, 1976.

[Strachey 67] Strachey, C.
Fundamental Concepts in Programming Languages.
Oxford University, 1967.

[Tanenbaum 78] Tanenbaum, A.S.
A comparison of Pascal and Algol68.
The Computer Journal 21(4):316-323, November, 1978.

[Taylor & Wilson 82]
Taylor, R. and Wilson, P.
OCCAM a new language for parallel processing systems.
Electronics (11), November, 1982.

[Teitelman 75] Teitelman, W.
INTERLISP Reference Manual.
Technical Report, Xerox Palo Alto Research Center, Palo Alto ,California, USA, 1975.

[Tennent 77] Tennent, R.D.
Language Design Methods Based on Semantic Principles.
Acta Informatica 8:97-112, 1977.

[Turner, D.A. 81]
Turner, D.A.
The Semantic Elegance of Applicative Languages.
In *Proceedings 1981 conference on Functional Programming Languages & Computer Architecture, Portsmith, New Hampshire.*, pages 18-22. October, 1981.

[van Wijngaarden *et al.* 69]
van Wijngaarden, A. *et al.*.
Report on the Algorithmic Language Algol 68.
Numerische Mathematik 14:79-218, 1969.

[Wasserman *et al.* 81]
Wasserman, A.I., Shertz, D.D., Kersten, M.L., Reit, R.P., and van de Dippe, M.D.
Revised Report on the Programming Language PLAIN.
ACM SIGPLAN Notices , 1981.

[Wirth 74] Wirth, N.
On the Design of Programming Languages.
North-Holland, Amsterdam, 1974, pages 386-393.

[Wirth 83] Wirth, N.
Texts and Monographs in Computer Science: Programming in Modula-2: Second Edition.
Springer-Verlag, Berlin, 1983.

[Wiseman & Hiles 68]
Wiseman, N.E. and Hiles, J.O.
A ring structure processor for a small computer.
The Computer Journal 10(4):338-346, February, 1968.