This paper should be referenced as:

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

An approach to persistent programming

M.P.Atkinson[+], P.J.Bailey[*], K.J.Chisholm[+], W.P.Cockshott[+] and R.Morrison[*]

[+]Department of Computer Science, University of Edinburgh,
Mayfield Rd., Edinburgh EH9 3JZ

*Department of Computational Science, University of St.Andrews
North Haugh, St.Andrews KY16 8SX

Abstract

This paper presents the identification of a new programming language concept and reports our initial investigations of its utility. The concept is to identify persistence as an orthogonal property of data, independent of data type and the way in which data is manipulated. This is expressed by the principle that all data objects independent of their data type should have the same rights to persistence or transience. We expect to achieve persistent independent programming, so that the same code is applicable to data of any persistence. We have designed a language PS-algol by utilising these ideas and constructed a number of implementations. The experience gained is reported here, as a step in the task of achieving languages with proper accommodation for persistent programming.

Introduction

The long term storage of data has been of concern to programming language designers for some time. Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation then some file I/O or database management system interface is used. Two views of data evolve from this. Data can either be classed as short term data and would be manipulated by the programming language facilities or data would be long term data in which case it would be manipulated by the file system or the database management system (DBMS). The mapping between the two types of data is usually done in part by the file system or the DBMS and in part by explicit user translation code which has to be written and included in each program.

These different views of data are highlighted when the data structuring facilities of programming languages and database management systems are compared. Database systems have developed relational, hierarchical, network and functional data models [20,11,19,30,27] whereas programming languages may have arrays, records, sets, monitors [13] and abstract data types [18].

That there are two different views of data has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total [14], concerned with transferring data to and from files or a DBMS. Much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. This is unsatisfactory because of the time taken in writing and executing this mapping code and also because the quality of the application programs may be impaired by the mapping. Frequently the programmer is distracted from his task by the difficulties of understanding and managing the mapping. The translation merely to gain access to long term data should be differentiated from translations from a form appropriate to one use of the data to a form suitable for some other algorithms. Such translations are justified when the two forms cannot coexist and there is a substantial use of both forms. The second major disadvantage is that the data type protection offered by the programming language on its data is often lost across the mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

We seek to eliminate the differences between the DBMS and programming language models of data. This can be done by separating the issue of what data structures are best from the issue of identifying and managing a property of data we call persistence. This is the period of time for which the data exists and is useable. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.
2. local variables in procedure activations.
3. own variables, global variables and heap items whose extent is different from their scope.
4. data that exists between executions of a program.
5. data that exists between various versions of a program.
6. data that outlives the program.

The first three persistence categories are usually supported by programming languages and the second three categories by a DBMS, whereas filing systems are predominantly used for categories 4 and 5. We report here on the PS-algol system which is a step in the search for language and database constructs which meet the needs of persistent data and hence obviate the need for the programmer to cope with the problems described above. PS-algol uniformly supports programming for an increased range of persistence. The system is implemented and is being actively used in a number of projects. Here we will describe the language design decisions in implementing persistence, the underlying implementation problems, the results obtained and some thoughts for the future.

The language design method

As a first attempt at producing a system to support persistence we hypothesised that it should be possible to add persistence to an existing language with minimal change to the language. Thus the programmer would be faced with the normal task of mastering the programming language but would have the facility of persistence for little or no extra effort.

The language chosen for this was S-algol [21,12], a high level algol used for teaching at the University of St Andrews. This decision was made by the University of Edinburgh team after some trouble with attempts at Algol 68 [34] and Pascal [36] and resulted in the two teams collaborating on the project.

S-algol stands somewhere between Algol W [35] and Algol 68. It was designed using three principles first outlined by Strachey [29] and Landin[17]. These are

* The principle of correspondence
* The principle of abstraction
* The principle of data type completeness

The application of the three principles in designing S-algol is described elsewhere [22]. The result is an orthogonal language whose "power is gained from simplicity and its simplicity from generality" [33]. Here we are interested in data and the S-algol universe of discourse can be defined by

1. The scalar data types are integer, real, boolean, string, picture and file.
2. For any data type T, *T is the data type of a vector with elements of type T.

3. The data type pointer comprises a structure with any number of fields, and any data type in each field.

The world of data objects can be formed by the closure of rule 1 under the recursive application of rules 2 and 3.

The unusual features of the S-algol universe of discourse are that it has strings as a simple data type [24], pictures as compound data objects [23] and run time checking of structure classes. The picture facility allows the user to produce line drawings in an infinite two dimensional space. It also offers a picture building facility in which the relationship between different sub-pictures is specified by mathematical transformations. A basic set of picture manipulating facilities along with a set of physical drawing attributes for each device is defined. A pointer may roam freely over the world of structures. That is a pointer is not bound to a structure class. However when a pointer is dereferenced using a structure field name, a run time check occurs to ensure the pointer is pointing at a structure with the appropriate field.

Together with our hypothesis of minimal change to the programming language we recognise certain principles for persistent data.

- **persistence independence** : the persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence and at other times only transient.

- **persistence data type orthogonality** : In line with the principle of data type completeness all data objects should be allowed the full range of persistence.

- The choice of how to provide and identify persistence at a language level is independent of the choice of data objects in the language.

A number of methods were investigated to identify persistence of data. Some involved associating persistence with the variable name or the type in the declaration. Under the rule of persistence independence these were disallowed. S-algol itself helped to provide the solution. Its data structures already have some limited notion of persistence in that the scope and extent of these objects need not be the same. Such structures are of course heap items and their useability depends on the availability of legal names.

This limited persistence was extended to allow structures to persist beyond the activation of the program. Their use is protected by the fact that the structure accesses are already dynamically checked in S-algol. Thus we have achieved persistence and retained the protection mechanism. Of course, in implementing this we had to devise a method of storing and retrieving persistent data as well as a description of its type and a method of checking type equivalence when data is reused in different programs. This was not a trivial task.

The choice of which data items persist beyond the lifetime of a program was next. We argued, by preaching minimum change, that the system should decide automatically. Such decisions are already taken in a number of languages like S-algol when garbage collection is involved and we therefore felt that reachability, as in garbage collection, was a reasonable choice for identifying persistent objects. However a new origin for the transitive closure of references, under explicit user control, which differentiates persistent data and transient data is introduced. Thus when a transaction is committed we can identify a root object from which we can find all the persistent data in the program. This data we preserve for later use.

PS-algol

Given the constraint of minimal change to S-algol the simplest way to extend the facilities of the language is by adding standard functions and PS-algol is implemented as a number of functional extensions to S-algol. In this way the language itself does not change to accommodate persistence. Thus the population of S-algol programmers could now use PS-algol with very little change to their programming style. In fact an S-algol program will run correctly with little diminution of speed in the PS-algol run time environment.

The functions added to support persistence are

procedure open.database (**string** database.name, password, mode -> **pntr**)

This procedure attempts to open the database specified by database.name (which in general is a path down a tree of directories, terminating in the database's name) in the mode (read or write) specified by mode, quoting password to establish this right. If the open fails the result is a pointer to an error record. If it succeeds the result is a pointer to a table (see below) which contains a set of name-value pairs and is the root from which preserved data is identified by transitive closure of reachability. A table is chosen so as to permit programs which use one route to the data to be independent of programs using other routes. Many databases may be open for reading, but only one may be open for writing so that the destination for new persistent data may be deduced. If this is the first successful open.database then a transaction is started.

procedure commit

This procedure commits the changes made so far to the databases open for writing. The program may continue to make further changes. Only changes made before the last commit will be recorded, so that not performing a commit is equivalent to aborting the transaction.

procedure close.database (**string** database.name)

This procedure closes the specified database making it available for other users who wish to write to it (multiple readers are allowed). It takes effect immediately. An implicit close database is applied to every database open at the end of the program. When a database has been closed references to objects not yet imported may remain accessible as may any imported data. An error is detected if the programmer tries to access any data that has not been imported from the closed database. A program may reopen a database it has closed but to avoid inconsistency between data that was held in the program and the database, it may only do so if there has not been an intervening write to that database.

These three procedures are those concerned with managing persistence, but there are also a set of procedures to manipulate tables, a mechanism for associative lookup, implemented as B-trees[8] or equivalent algorithms. A table is an ordered set of pairs. Each pair consists of a key and a value. At present the key may be an integer or string value, and the value is a pointer to a structure instance or table. The basic library for tables is:

procedure table (-> **pntr**)

This procedure creates an empty table represented by an instance of the structure class Table.

procedure s.lookup (**string** key ; **pntr** table -> **pntr**)
procedure i.lookup (**int** key ; **pntr** table -> **pntr**)

These procedures take a given key and considering only the pairs within the table return the last value stored by a call of s.enter or i.enter for this key, or nil if there is no such pair.

procedure s.enter (**string** key ; **pntr** table, value)
procedure i.enter (**int** key ; **pntr** table, value)

These procedures store in the given table a pair or if the supplied value is nil delete the previously stored value for the given key.

procedure s.scan (**pntr** table,environment ; (**string**,**pntr**,**pntr** -> **bool**) user -> **int**)
procedure i.scan (**pntr** table,environment ; (**int**,**pntr**,**pntr** -> **bool**) user -> **int**)

These procedures provide iteration over tables. The function user is applied to every pair with a key of the appropriate type, in ascending order of integers or lexical order of strings, until either it yields false or the whole table has been scanned. The first parameter supplied to the function user is the key, the second is the corresponding value and the third parameter the value given as environment. The result of these scan procedures is the number of times user is applied. An example of their use is given in figure 3.

procedure cardinality (**pntr** table -> **int**)

This procedure returns the current number of entries in the given table.

The table facilities were envisaged as a way of packaging index constructions. For example singly and multiple indexed relations can readily be constructed using them. They have also proved very popular as a dynamic data structure constructor. We present as an example of their use three PS-algol programs to maintain a simple address list. The program in figure 1 adds an new person, that in figure 2 looks up the telephone number of a person and that in figure 3 finds the longest telephone number.

```
structure person (string name, phone.no ; pntr addr, other)
structure address (int no ; string street, town ; pntr next.addr)
let db = open.database ("Address.list", "Morwenna", "write")
if db is error.record then write "Can't open database" else
begin
        write "Name : "             ; let this.name = read.a.line
        write "Phone number : "     ; let this.phone = read.a.line
        write "House number : "     ; let this.house = readi
        write "Street : "           ; let this.street= read.a.line
        write "Town : "             ; let this.town = read.a.line
        let p = person( this.name,this.phone, address (this.house, this.street,
                        this.town, nil), nil)
        let addr.list = s.lookup ("addr.list.by.name", db)
        s.enter (this.name, addr.list,p)
        commit
end
```

A program to add one new person to a database containing an address list
Figure 1

```
structure person (string name, phone.no ; pntr addr, other)
let db = open.database ("Address.list", "Kernow", "read")
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup ("addr.list.by.name", db)
write "Name : " ; let this.name = read.a.line
let this.person = s.lookup (this.name, addr.list)
if this.person = nil then write "Person not known" else
        write "Phone number is : ", this.person (phone.no)
```

A program to look up the telephone number of one person from the address list
Figure 2

```
structure person (string name, phone.no ; pntr addr, other)
let db = open.database ("Address.list", "Kernow", "read")
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup ("addr.list.by.name",db)
structure env (int max ; string longest)

let e1 = env (0,"")
procedure phone (string name ; pntr val, e1 -> bool)
begin
        let number = val (phone.no)
        let len = length (number)
        if len > e1 (max) do { e1 (max) := len ; e1 (longest) := number }
        true
end

let count = s.scan (addr.list, e1, phone)
if count = 0 then write "Nobody in the list yet" else
        write "Longest telephone number is : ", e1 (longest)
```

A program to find the longest telephone number in the address list
Figure 3

Note that in these examples the declared data structures are different. This identifies which parts of the data each program may touch and so behaves like a simple database view mechanism. Only those quoted will be matched against the stored set of data descriptions when they are first used. Paths to other data, eg alternative addresses via next.addr and extra information about people via other have not been used. They are there to give access to data pertinent to other programs or to future needs and they do not clutter programs which do not use them. Similarly the standard table is used to name access paths and only those used need be considered in a particular program. Figure 4 shows a program to introduce a new access path to the address list. With these disciplines, which we find useful, data design and programs can grow together. These examples illustrate the ease with which small programs operating on persistent data may be composed to provide a complete system. In this example one would imagine other programs: to start an address list, to delete it, to remove entries, amend addresses, print lists etc. We believe this method of constructing software tools to be an attractive way of building such systems.

As well as showing these features of persistence the programs illustrate features of the parent language S-algol. For example, the type of a name is deduced from the initialising expression, this leads to conciseness. The freedom to place declarations where they are needed means that constant names are common and programs may be read without searching far for declarations.

```
structure person (string name, phone.no ; pntr addr, other)
let db = open.database ("Address.list", "Morwenna", "write")
if db is error.record do { write "Can't open database" ; abort }
let addr.list = s.lookup ("addr.list.by.name", db)
let phone.number.table = table !create a new empty table

procedure put.it.in.phone.number.table (string name;pntr val, env -> bool)
        { s.enter (val (phone.no), phone.number.table,val) ; true }

let count = s.scan (addr.list, nil, put.it.in.phone.number.table)
s.enter ("addr.list.by.phone.number", db, phone.number.table)
commit
```

A program to construct a new index onto the address list, by phone number
Figure 4

The persistent object management system

The persistent object management system is concerned with the movement of data between main store and backing store. This is controlled by the run time support system where transactions are implemented by Challis' algorithm [10]. The movement of data is achieved as follows.

Movement of data on to the heap

Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data object. The second mechanism is invoked when a pointer being dereferenced is a **p**ersistent **id**entifier (**PID**). The persistent object manager is called to locate the object and place it on the heap, possibly carrying out minor translations. The initial pointer which is a **PID** is that yielded by open.database and subsequent pointers will be found in the fields of structures reached from that reference.

Movement from the heap

When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk. Some anticipation of these transfers may be necessary if the heap space is insufficient for the whole transaction.

The algorithms and data structures used to implement this data movement are described elsewhere [3,4]. Since the database may be shared by many programs the binding of the names of persistent data must by dynamic and symbolic. In PS-algol this binding is performed automatically when the object is accessed for the first time. There is no overhead in accessing local objects.

Type checking is also performed by the system. Remember that a pointer may roam over the domain of structure classes. Thus when a pointer is dereferenced to yield a value the system must check that the pointer points to a structure with the correct field name. The field name and the structure incarnation must carry around type information to enable this checking to be performed.

To extend this type checking to persistent structures it is sufficient to ensure that the type information migrates with the structure itself. This is accomplished by making the type information an implicit field of the structure thus guaranteeing that the type information will persist.

Design issues in persistent object management

There are a number of tradeoffs to consider when designing the persistent object manager used to implement the data migration. Some of these are illustrated here. We may choose between making all references suitable as disc references (as in Smalltalk [16]), translating every time or we may (and usually do) economise by storing a mapping and translating less often. Then a choice exists as to when to build up the map. Do we make an entry when a reference is first introduced or when it is first dereferenced? Similarly do we make all references go via this map? There are problems of how to store the map so that it grows on demand rather than causing a high initial overhead. We require access to the map to be fast in both directions. These decisions interact with store management. We wish to avoid putting data on disc if we can determine it is not reachable. We may use these transfer mechanisms to avoid space exhaustion after garbage collection. When virtual memory [31] is available it interacts with these algorithms and may be exploited if the operating system permits. On the other hand we would like a run time system which is portable.

The design of the address structure for the persistent data allows interaction between addressability and the cost of disc garbage collection. This influences attempts to make the address structure extensible, to make it space efficient and to make it address sufficient data. Placement strategies, data compression techniques and variations on the method of implementing transactions again interact and have significant effect on performance. The choice of a particular implementation is therefore a choice of a particular point in an extensive and many dimensional space. Our present choice described below is empirical and pragmatic rather than being based on systematic evaluation of this space or the optimisation of some abstract model.

Thus venturing into this approach to providing persistence has provoked many research issues concerning implementation which we have only just begun to explore and which will be the topics of other papers, the first of which is in preparation [5].

An implementation of persistence

Our earliest implementation of persistence used a linear table of PIDS and local addresses, with entries for every pointer in imported objects (objects that had been brought in from the database). Hashing was used to accelerate lookups of a PID (needed when a object is being imported to translate pointers to objects already imported). In the next implementation the position in this table was referred to as the local object number LON, and all references in all objects were represented by their LON. This cost an indirection in every reference (using an addressing mode on the VAX11) but meant that the local address to PID translation, needed on transferring data to the database (export) was merely indexing the table. Hash coding accelerated PID to LON translation and Bloom filters [9] were used to reduce the cost of discovering a PID had not yet been encountered. The main disadvantage of this was that an entry in the PIDLAM was needed for every active object so this table grew large. That implementation achieved good performance by using the virtual memory mechanisms of VAX, implementing Challis' algorithm as page tables stored at the start of the file and arranging that altered pages were paged to a new site.

Our present implementation, POMS (persistent object management system), is designed to overcome many of the deficiencies of the earlier systems, it is described in detail elsewhere [7]. Briefly, the design depends on adaptive structures for all the addressing mechanisms, to avoid the problems of a high fixed overhead, or a small limit to the maximum volume of data that can be accommodated. The PIDLAM is now two hashing structures which grow as necessary, one from PIDs to local addresses, the other from local addresses to PIDs, for only the pointers which have been both imported and dereferenced. Translation is done when a reference is first used to minimise the size of these tables.

To avoid a large number of transfers on startup, the logical to physical disc address mapping is itself mapped, using its own mapping and a bootstrap. So this map may grow as the database grows and is loaded incrementally. To avoid dependence on particular operating system features and the problem found when implementing transactions that file operations and database operations can get out of step, we include our own directory mechanism for databases. Depending on this strategy and writing most of the code for POMS in PS-algol itself, we believe we have achieved reasonably portability.

The objects are clustered by type in the database pages, so that the type description information may be factored out. The order of transfers during import is not under POMs control since it depends on the order the program accesses structures. During the routines for commit, when data is exported, the total set of data which cannot become reachable from the root is identified from local information and discarded. The imported objects that have been changed and the newly created objets reachable from them, then have to be exported. This list of exports is sorted to minimise transfers and head movement, before write-back. The combination of these tactics, which we continue to refine, gives reasonable performance. Better performance may be achievable by exploiting segmentation hardware within the operating system.

Experience using PS-algol

At first sight the set of facilities provided by PS-algol may look fairly primitive. Notice however that the programmer never explicitly organises data movement but that it occurs automatically when data is used. Notice also that the language type rules are strictly enforced and that the programmer uses a method already familiar to him to preserve data. That is by the usual naming convention where the preservation of data is a consequence of arranging that there is a way of using the data. Thus we have achieved persistence by minimal change allowing the programmer to use all his familiar techniques of problem solving.

The effect on programs written in PS-algol has been quite dramatic. We have some early results of tests comparing programs written in PS-algol with programs written in Pascal with explicit database calls. These programs implemented a DAPLEX [27] look-alike, a relational algebra and various CAD and demonstration programs. We have found that there is a reduction by a factor of about three in the length of the source code. These are of course early results and will need further confirmation. However it is the sort of result we expected.

We have also found that the coding time for these programs is reduced by at least the same factor. We suspect that the maintenance of the programs will be easier. We avoid the layering costs associated with calls to successive levels of a DBMS as data is brought into the normal program from the heap. Consequently we have observed a reduction in CPU requirements for equivalent programs even though we are still interpreting. Whether there is an overall increase in speed depends on the data structures and the algorithms over them and we hope to investigate performance further.

Evaluation of persistence as an abstraction

The abstraction of persistence has been so successful that we would recommend that other language designers consider it for the languages they design. It identifies significant aspects of common programming tasks consequently reducing the effort required by the programmer to accomplish those tasks. It abstracts away much detail commonly visible to the programmer so that programs produced using it are simpler to understand and to transport. It is feasible to implement.

Persistence has also appeared as an orthogonal property of data in the work of Albano et al[1]. In an attempt to accommodate longer term persistence they make contexts proper objects which can be manipulated, as we suggest for an identified subset of contexts in our language proposals NEPAL[2]. In another group of languages PASCAL/R[26], RIGEL[25] & PLAIN [32] the designers have chosen not to adopt the principle of data type completeness, and have only allowed instances of type relation to have longer term persistence. It is interesting to note that in PASCAL/R the construct DATABASE has a form like a PASCAL record, and if its fields were allowed to take any of the PASCAL data types then that language would be consistent in allowing data structures of any type to have any persistence. ADAPLEX[28] is constructed by merging a given database model, DAPLEX[27] and an existing language ADA[15], with the inevitable consequence of restrictions on which data types can have which persistence. When casting program examples to assess languages [6] we have found it particularly irksome if data types which can persist cannot also have temporary instances. We conclude, therefore, that the consistency of our form of persistence is of advantage to the programmer.

Conclusions

The abstraction achieved by treating persistence as an orthogonal property of data has been shown to have many interesting properties. It is clear it favourably affects program length, program development time and program maintainability. Demonstration implementations of a particular flavour of this idea have shown it to be practicable for reasonable amounts of data. This particular flavour does not have concurrency other than at database level. This may be a limitation which prevents its application to simple transactions on large database systems but

this leaves the very substantial number of programs which run against conventional files which implement complex interactions as in CAD, or use personal databases as uses of this language.

Acknowledgements

The work at Edinburgh was supported in part by SERC grant GRA 86541. It is now supported at Edinburgh by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by grants from ICL.

References

1. Albano, A., Cardelli, L. & Orsini, R. (1982). Galileo : a strongly typed interactive conceptual language.

2. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1982). NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9,8 299-318.

3. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1983). CMS : A chunk management system. accepted for publication Software, Practice & Experience

4. Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. & Marshall, R.M. (1983). Algorithms for a persistent heap. accepted for publication Software, Practice & Experience

5. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. (1983). An exploration of various strategies for implementing persistence as an orthogonal property of data. in preparation.

6. Atkinson, M.P. & Buneman, P. (1983). Survey paper on persistent languages. in preparation.

7. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. (1983). The persistent object management system. in preparation.

8. Bayer, B. & McCreight, A. (1972). Organisation and maintenance of large ordered indexes. Acta Informatica 1 173-189.

9. Bloom, B.H. (1970). Space/time tradeoffs in hash coding with allowable errors. Comm.ACM 13,7 422-426.

10. Challis, M.P. (1978). Data Consistency and integrity in a multi-user environment. In Databases : improving usability and responsiveness Academic Press 245-270.

11. Codd, E.F. (1970). A relational model for large shared databases. Comm.ACM 13,6 377-387.

12. Cole, A.J. & Morrison, R. (1982). An introduction to programming with S-algol. Cambridge University Press.

13. Hoare, C.A.R. (1974). Monitors : an operating system structuring concept. Comm.ACM 17,10 549-557.

14. IBM report (1978). on the contents of a sample of programs surveyed. San Jose, California

15. Ichbiah et al (1979). Rationale of the design of the programming language Ada. ACM Sigplan Notices 14,6.

16. Kaehler, T. (1982). Virtual memory for an object-orientated language. Byte 378-387.

17. Landin, P.J. (1966). The next 700 programming languages. Comm.ACM 9,3 157-164.

18. Liskov, B.H. et al (1977). Abstraction mechanisms in CLU. Comm.ACM 20,8 564-576.

19. Lochovsky, F.H. & Tsichritizis, D.C. (1978). Hierarchical database management systems. ACM Computing Surveys 8,1 105-123.

20. Lochovsky, F.H. & Tsichritizis, D.C. (1982). Data Models.

21. Morrison, R. (1979). S-algol language reference manual. University of St Andrews CS/79/1.

22. Morrison R. (1982). Towards simpler programming languages : S-algol IUCC Bulletin 4,3 (October 1982)

23. Morrison, R. (1982). Low cost computer graphics for micro computers. Software, Practice & Experience 12 767-776.

24. Morrison, R. (1982). The string as a simple data type. ACM.Sigplan Notices Vol 17,3.

25. Rowe L.A. Reference manual for the programming language RIGEL.

26. Schmidt, J.W. (1977). Some high level language constructs for data of type relation . ACM.TODS 2,3 247-261.

27. Shipman, D.W. (1981). The functional data model and the data language DAPLEX. ACM.TODS 6,1 140-173.

28. Smith, J.M., Fox, S. & Landers, T. (1981). Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts

29. Strachey, C. (1967). Fundamental concepts in programming languages. Oxford University Press.

30. Taylor, R.C & Frank, R.L. (1976). CODASYL database management systems. ACM Computing Surveys 8,1 67-103.

31. Traiger, I.L. (1982). Virtual memory management for database systems. ACM.Sogops 16,4 26-48.

32. Wasserman, A.I., Sheretz, D.D., Kersten,M.L., van de Reit, R.D. (1981). Revised report on the programming language PLAIN. ACM.Sigplan Notices 16,5.

33. van Wijngaarden, A. (1963). Generalised algol. Annual Review of automatic programming 3 17-26.

34. van Wijngaarden, A. et al (1969). Report on the algorithmic language Algol 68. Numerische Mathematik 14 79-218.

35. Wirth, N. & Hoare, C.A.R. (1966). A contribution to the development of algol. Comm.ACM 9,6 413-431.

36. Wirth, N. (1971). The programming language Pascal. Acta Informatica 1 35-63.