WHAT'S WRONG WITH APL?

Philip S. Abrams

Scientific Time Sharing Corporation
7316 Wisconsin Avenue
Bethesda, MD 20014 USA

APOLOGIA

Throughout history, every time a new idea has come along there have been many people quick to criticize it. As often as not, such criticism has come from detractors of the idea, and has been motivated by its threat to older, more established beliefs. The Biblical prophets, Socrates, Jesus, Copernicus, Galileo, Pasteur, Marx, Darwin, Stravinsky, and countless others, all experienced resistance to their ideas for essentially emotional rather than intellectual reasons.

From its early days as "Iverson Notation" through its more recent development, APL has been the target of heated discussion. This paper is a criticism of APL, but I believe, different from others. I am not a detractor of APL; in fact, I have been a supporter, developer, and promoter of the language for quite some time. Therefore, the intention of this review is not to suggest that since APL has faults it is worthless. To the contrary, I hope that these comments will lead to further improvements of APL and perhaps suggest some of the directions to consider in the development of its successors.

This paper could not have been written much earlier. It is because APL has come of age, both in the theoretical domain and in the commercial world, that it is possible to look at it publicly with a critical eye.

The discussion that follows is written for the APL community, present and future. My wish is that APL "believers" will accept this analysis in the constructive spirit in which it is offered, and that those who still do not appreciate the beauty, elegance, and practical power of APL will not take these comments out of the context in which they are presented.

INTRODUCTION

We intuitively expect that in a "good" programming language, the complexity of a program should be closely related to the difficulty of the problem it solves. While this is generally the case in APL, one often encounters relatively simple problems whose APL solution is unusually awkward or complicated. In most such cases, the source of difficulty is an inability to express some fundamental concept or structure in a natural way. Experience with other programming languages effectively demonstrates the value of constructs and facilities that APL does not have.

There is a fine distinction between capabilities that are desirable but missing, and those that are really essential. Since APL can specify any computable function, one could argue that nothing is missing from the language. However, by the same argument, any computable function can also be expressed in FORTRAN or by some Turing machine, so APL is not needed at all! In general the approach in this paper is pragmatic. I consider those areas "trouble spots" where there is a large distance from how I think of a problem to how I must write it in APL.

My goal in this paper is to catalog these areas of difficulty in APL. Though some of my preferences may be apparent, it is not my intention to make specific proposals here. I do, however, suggest that the topics considered in this paper deserve further serious study.

To discuss APL's problems, we must agree upon what APL is. For the purpose of this paper, APL is taken to mean "those ideas and concepts embodied in the language implemented by APL\360 and APLSV" (viz. Gilman and Rose 1974). Note the emphases in the previous sentence. I intend not to consider those aspects of the above-named systems that are clearly implementation limitations or bugs. For instance, although $A[A]\leftarrow V$ invokes an error report in APL\360, it is part of APL as defined above.

In this paper, the primary focus is the language and not the systems that support it. This is a bit of a fuzzy area, as some of what is done by system

commands or system functions ought perhaps to be included in the language proper, as we shall discuss later.

For the purpose of exposition, problem areas are classified as follows:

Data structures
Control structures
Naming, binding, and security
Syntax
Primitives

These areas are by no means orthogonal. Many problems of different types are interrelated, and it is likely that a solution to one might indeed include solutions in other areas.

DATA STRUCTURES

Semantically the biggest contribution of APL to computer science is its superb treatment of arrays. A simple and inclusive definition of rectangular arrays allows all such objects from scalars through higher ranks to be treated consistently. A harmonious set of primitive functions and operators makes it possible to express most transformations of array-structured data cleanly, concisely, and unambiguously. The relative ease of treatment of the semantics of arrays and their associated primitives reinforces the belief that the current formulation is a "right" one (e.g. Abrams 1970, Brown 1971, Gerhart 1972, More 1973).

APL's ability to deal with rectangular homogeneous arrays is unsurpassed. Unfortunately, not all data are most naturally represented in this form. "Real world" problems often require data structures that are nonrectangular as well as nonhomogeneous. For example, although a list of names can be represented as a character matrix, a vector of character vectors might be a more convenient representation.

A definite need exists for tree-structured data. There have been numerous proposals over the years for extensions to generalize arrays in this direction (e.g. Abrams 1966, Brown 1971, Edwards 1973, Ghandour and Mezei 1973, Murray 1973). My current feeling is that these proposals are too complicated. Most require a lot of extra machinery in APL to make the tree structured extensions work right. Many of these proposals seem to founder in their treatment of "scalars" -- that is, atomic objects with no associated structure. The complexities and the deficiencies in limiting cases strengthen my belief that no one has yet made the same leap of genius with respect to generalized structures that Iverson made for rectangular arrays.

The problem of defining the new structures is the same as that for APL arrays: to find a simple structure and the appropriate primitive functions to deal with it naturally. It is not necessary that these new structures be extensions or generalizations of APL arrays. Indeed the correct structure may bear no relation to arrays as we know them now in APL.

It has often been suggested that numbers in APL be considered to be embedded in the complex field rather than in the field of reals. Such an extension would, of course, imply similar extension to primitive functions (McDonnell 1973). Now new problems arise for users who are interested only in reals and who may not want their results to pop up in the complex field. What is the best way to provide new types or extend old types without forcing them on everybody and every application?

Other examples of lacking structures are plentiful: sparse arrays, unordered sets, nonhomogeneous arrays, lists, files, graphs, functions, expressions. Is the true need in APL for a few more primitive data structures? Or is it rather for an extensional capability allowing users to define their own structures?

A defender of the sufficiency of APL's data structures would argue that most of the structures discussed above can easily be represented by rectangular arrays in APL. In practice, simulating a structure in APL often leads to dismaying complexity. Suppose, for instance, that we wish to deal with LISP-like structures and primitives. One approach is to write a complete LISP system, including statement interpretation, storage, and name management. Here, all input is via "quote quad" and I need not even be aware that the LISP system is written in APL. A second approach is to model the LISP structures as APL arrays and to build APL functions corresponding to the primitives of LISP. In this case, I have APL functions with names like *CAR* and *CDR*, which I employ within APL expressions.

Neither of these methods is wholly satisfactory. The first is preferable in that the final result behaves exactly as desired. This, however, is at the price of a lot of work and a lot of machinery which duplicates functions inherent to an APL system. With this approach, it is impossible to access both the APL and the LISP data in a single expression. The name spaces of the two are so isolated that it is necessary to switch contexts to get at them both.

On the other hand, the second approach, while simpler, engenders problems of name conflicts between the user's identifiers and those identifiers used in the modeled primitives. Worse, there is no simple way to distinguish objects of the new data type from ordinary

APL arrays. This latter is, of course, an advantage to the simple creation of such systems, but provides numerous pitfalls to even the sophisticated user.

In fact, it would be nice if we didn't have to simulate LISP structures at all. What are the properties of LISP data and primitives that allow them to express certain algorithms more easily than APL? How, if at all, might these characteristics be integrated into APL?

CONTROL STRUCTURES

It is amazing that APL, which is so rich in data-handling primitive functions, should be so poor in program structures. The user can create new functions on data to handle specialized tasks not done directly by the primitives. This is not possible in the area of control. As with data structures, the fundamental problem is not the ability to express some algorithm, but the faculty to do so naturally.

Branching. Simple branching to line numbers is unusually crude. The APL\360 branch is a step backwards from Iverson Notation, in that the early branch arrows indicated program structure quite clearly (see Iverson 1962, Falkoff et al. 1964). This aspect of the earlier two-dimensional notation was lost with the linearization of APL. Unlike other primitive functions and operators, branching is neither intuitive nor natural to humans.

The rigidity of APL's control mechanisms stands in particularly sharp contrast to its flexibility in dealing with data. Programmer-defined functions, with the same syntax as primitive functions, extend data-manipulation abilities to an extent limited only by the skill and imagination of the programmer. But how does one use the branch arrow as a primitive to build more elaborate or more congenial control structures?

Some structures, like the BASIC for statement, can almost be achieved; others, such as an interrupt mechanism or an exit to a specified point in a calling program, just can't be done at all. Perhaps the most startling instance of APL's weakness in control structures is that it is impossible to write a function *GOTO* , whose behavior is identical to the primitive branch, → . Those control structures that can be built are "fragile", in that they depend heavily on the programmer's cooperation and so are really usable only by the originator in "toy" applications.

The use of line numbers as branch targets poses a serious threat to the integrity and thus the reliability of a program. (The catastrophic consequences of errors in calculated branches are well-known to all programmers!) Likewise, the possibility of branching to any line of a function destroys all hope of protecting loops or critical sections from undisciplined entry. Computed goto's also complicate program optimization.

The use of line numbers, rather than a more complete and consistent naming convention for program points, creates a confusion between lines of a function (an artifact of typewriter terminal usage) and statements. APL\360 has exactly one statement per line; it is often desirable to allow groups of statements on the same line as well as to permit a statement to occupy several lines. There is no good reason to confuse typography with syntax.

Subroutine Call. The limitation of functions to valence ("adicity") 0, 1, and 2 has severe effects. Of course, functions with three or more scalar arguments can be written to use a single vector argument, but this trick fails as soon as one or more of the arguments is a vector or array. While there are various other artifices for simulation of multivalence, such as the use of global variables or descriptors, each requires the programmer to depart from normal function syntax (e.g. Rose 1971). A consequence of the valence restriction is the loss of structure in systems of programs. Such approaches also make it more difficult to name actual parameters to a function.

Defined functions cannot be of variable valence. This facility is available to primitive functions, such as + , but not to user-defined functions. The lack of this facility makes it impossible to write APL functions that model APL primitives, which is at least an embarrassment.

Along with the need for more than two arguments, some functions such as domino (⌹) ought perhaps to return multiple results. Domino now loses half of its potential useful results -- the residuals calculated as part of the solution to a linear system. There seem to be two options, equally unattractive. With the first, part of the result would be stored as a global or system variable, that is destroyed by each subsequent call of domino. With the second, the function would return a complex structure as a result. This is just as bad; the result, to be useful, must still be dismembered. This may be an area in which there is no "APLish" solution.

Many problems require call by reference or the equivalent; that is, it should be possible to pass an unevaluated but bound name as an actual parameter. The "execute" function does not satisfy this need, as we will discuss later.

One of the essential requirements of the stepwise refinement or top-down

programming technique is that, within any program or system of programs, all subprograms may be viewed as "black boxes". The choice of local names, subprograms, programming techniques, and so forth within a box should not be constrained by choices made within higher level programs, and vice versa. One adverse consequence of APL's elegant name-localizing rule is that the black-box approach cannot generally be realized, because names cannot be made "strictly local" to a program.

Although it hampers subprogram isolation, APL's localization rule provides flexibility in communication between calling and called programs that is unequaled in most other programming languages. Further, because the names accessible by a program are exactly those accessible at the point of call, APL has the significant advantage of allowing subprograms to be tested and observed in vitro.

The only control mechanism for getting out of a function is a return to the point of call. In addition to this ability, it is sometimes necessary to perform uplevel transfers of control to a specified point in the global environment, as for error returns. Similarly, no "sideways" control transfer exists; that is, the subroutine call and return are not sufficient to model parallel or cooperating processes. This kind of facility is found in simulation languages in the form of coroutines.

Ever since Dijkstra denounced the goto, "structured programming" has become a popular addition to the jargon of programmers and language designers. As mentioned earlier, the simple goto of APL is insufficient, and there is a clear need for at least statement grouping, alternative choice, and repetition. Numerous proposals (e.g. Kelley 1973, Harris 1973) have suggested adding what amounts to ALGOL 60 control primitives to APL. While I applaud the spirit in which these extensions have been offered, I believe that control primitives appropriate to a "scalar" language like ALGOL or PL/I are not necessarily those best suited to APL. Indeed this area, like data structures, is probably awaiting another revolutionary jump to provide the right solution.

One of the most serious practical problems in construction of real systems and packages is the need to monitor and control the environment, notably for error conditions and external events. Shared Variables (Falkoff and Iverson 1973) appear to offer a partial solution to a segment of the problem. It is eminently clear to application designers that some form of interrupt processing is required. This kind of facility would permit recognition of and reaction to synchronous events such as errors, as well as to asynchronous events such as terminal conditions, timers, the state of concurrent programs, or other occurrences "external" to the immediate sequential environment.

NAMING, BINDING, AND SECURITY

One of the most attractive features of APL is the elegance and simplicity of its naming and scope rules. Dynamic localization and the lack of a distinguished set of reserved words allow many programs to be relatively simple and straightforward. The addition of the quad-name conventions of APLSV allows more direct access and control of environmental and external variables and processes. Finally, because reference or label variables do not exist in the language, there are no lifetime or alias problems.

This simplicity, while convenient for small problems, seems to work against the creation of secure large-scale packages. By "security" I mean full protection from modification or subversion by either the untrained or the malicious user. The various limitations related to naming and binding are all reflected in the question of security.

There is no way to name, and therefore to access, shadowed variables and functions. Thus one cannot write a function to put temporarily unused values on an external medium, such as a file, in order to reduce space requirements. This inability would be even more strongly felt if an interrupt mechanism existed which could react to resource limitations of this sort.

Contrary to popular belief, the "execute" function (⍎) does not provide a reliable "call by name". What is missing is the ability to associate with a formal parameter the name bindings in effect at the point of call. Name localization can create anomalies when a name passed in a character string is shadowed, often inadvertently.

```
∇N IDENT X
 ⍝ DEFINE VARIABLE NAMED IN RIGHT
 ⍝ ARGUMENT AS IDENTITY MATRIX
  ⍎X,'←(⍳N)∘.=⍳N'
∇
```

called: `100 IDENT 'N'`

This name conflict can be circumvented only by enforcing naming conventions -- an approach that is impractical unless the program's only user is its creator.

A program's local identifiers cannot be made "invisible" to the user. Likewise, it is not possible to meld a program and its subprograms into an indivisible object. Other languages overcome these problems through use of compilers (which

purge most names from the object program) and loaders or linkage editors (which combine programs and eliminate most remaining names). The programmer can control the unification of a package by specifying entry points, external names, control sections, and so forth. APL's need for similar facilities has been recognized by others (Ryan 1973, Puckett 1974) and must be pursued further.

The extensions of APLSV allow parts of the environment to be localized within functions. In some cases, such as printer width, this is too restrictive, as the width is more properly associated with a terminal session than with a function or workspace. In other cases, notably index origin, the APLSV localization is too weak. Rather than treat index origin as a local variable which must be initialized at each execution, its value should be settable when its locality is declared. This approach would also eliminate the need for "IMPLICIT ERROR".

An explicit ability to deal with names and their values would allow rationalization of the workspace. With such functional capabilities in the language, a workspace becomes simply a universe of discourse -- that is, a set of identifiers bound with their values or definitions. What are now system-related mechanisms, like "LOAD" and "SAVE" would become simple APL functions to switch context and bind names to objects. Aside from simplification and unification, such mechanisms would make it possible to separate naming problems from storage considerations. They would also allow much of what is "mysterious" about APL systems to be described and extended in APL.

Naming and binding are closely tied to control structures. The inability to define lexically local functions or strictly local names are examples.

In summary, the problems with naming in APL seem to boil down to two essentials. First there is no explicit name type in APL. Since they are not part of the universe of discourse, names cannot be manipulated, or described, directly. One of the best examples of this is in the "Formal Description of APL" (Lathwell and Mezei 1971), where the authors were forced to ignore assignment, as such a description would require a way to discuss names and associated values. The second basic problem is that there is no way to control the scope of names, either in time (dynamic) or in context (lexical). Primitive facilities in this area would make "execute" more useful and would help in the rationalization of function definition. The ability to deal with names and scope would go a long way toward permitting the creation of secure functions and applications.

SYNTAX

Compared to other popular programming languages, APL has a simple and uniform syntax. Although rich in primitive functions, APL has simple precedence rules which are a boon to novice and experienced programmers alike. A straightforward approach to function precedence is imperative for APL, since the language has historically grown by extending the function set rather than by adding new syntactic statement types. Indeed there are but two types of executable statements in APL, in contrast to the large number found in other current languages.

Paradoxically, it is this very simplicity of APL syntax that stands in the way of well-designed extensions to the language. It is difficult or impossible to introduce new constructs and stay within APL syntax. Try, for example, to design extensions for functions of more than two arguments, or more than one result; or for operators or other new control structures; and retain consistency with existing syntax rules.

A new capability must either conform to the procrustean syntax of monadic and dyadic functions, or must escape from APL syntax entirely. One popular approach has been to enclose deviant material in quotes, as with the left arguments to formatting functions such as ΔFMT or α .

The function header is used to describe properties of a function -- its name, syntax, names of local identifiers -- that pertain throughout the function's execution. Elsewhere in this paper I have suggested the need to describe other properties of functions, such as index origin, strict locality of names, and multivalence. The current header syntax is inadequate for describing these additional properties.

The issue of function precedence in APL is the single most controversial, emotion-laden, and misunderstood aspect of the language. The common confusion of function precedence with order of execution has thoroughly muddled these two simple but essential notions. In the expression $A\times B+C$, function precedence indicates that A and $B+C$ are the operands of the function $\times$. An "order of execution" rule, for which this is commonly mistaken, would also require that $B+C$ be evaluated before A -- surely a curious requirement for $\times$, a commutative function!

APL's right-associativity rule is useful primarily because it tends to reduce the number of parentheses required to write an expression, although it does not necessarily minimize parentheses. Right rather than left associativity was chosen for APL because of the right

association of monadic functions with their arguments (Iverson 1966). Note, however, that left associativity might engender fewer parentheses. There is, however, nothing sacrosanct about associating arguments from only one end of an expression. For instance, one useful avenue to explore would be the use of short scope of right arguments, as described by More (1973). His dot notation, well-established in symbolic logic, might also be useful for minimizing parentheses in complex expressions.

If all APL functions simply transformed their arguments without modifying the global state, there would be no need to discuss order of evaluation. This order could then be any that respects the rules of association of functions with arguments. The existence of side effects (primarily through assignment) is what has made it imperative to consider order of execution of APL programs.

The most popular argument in favor of a specified order of execution, in particular "right to left", is that it allows the embedding of assignments to variables used "later" in an expression. On the other side, the cult of "one-liners" has demonstrated how easy it is to write incomprehensible expressions using internal assignment. Uncharitable critics as well as overzealous practitioners have concentrated overmuch on this potential of the language (e.g. Dijkstra 1972). Stylistically, this kind of programming is complex to read, to write, and to modify (Abrams 1973). Because of the side effects, such statements also tend to be difficult to debug, as they cannot be restarted. Having no special order of execution, beyond that required to respect association, would discourage this practice. Thus we would require evaluation of arguments before their passage to a function, but no further order would be specified.

The other principal argument against defining an order of evaluation is that it makes it much more difficult to write highly efficient APL processors. If the sole rule is to evaluate arguments to a function before invoking the function, the processor has more liberty to permute operation order and to do multiple operations in parallel. In the absence of side effects, there is no ambiguity.

PRIMITIVES

As I noted earlier, the main importance of APL is its development of the use of rectangular arrays through a complete and generally consistent set of primitive functions and operators. Their amenability to formal treatment suggests that the match between data structure and primitives is an unusually good one.

The definitions of some primitive functions are questionable. For instance, limiting cases of division and exponentiation (`0÷0` and `0*0`) do not agree with traditional mathematics. The inclusion of domino and the circular functions raises questions about what should and what should not be included in APL as primitive functions. Although useful in some application domains, these functions are relatively specialized and are much less fundamental than, say, indexing. Similarly, the ad hoc definition and scant powers of the ⍕ formatting function make its inclusion dubious. In the above cases, it might be more appropriate either to include more primitive constructs from which these functions might be defined conveniently, or to find more general functions from which these fall out as special cases.

Some functions are not yet general enough. Both expansion and "overtake" would be more useful if it were possible to specify an arbitrary fill sequence. When using expansion, one often regrets the lack of the "mesh" function of early versions of APL (Iverson 1962). The "take" function is overburdened in that it performs both selection and extension, depending on the value of its left argument. It might be preferable to separate the two functions.

There is a need for general string manipulation primitives. In particular, search and substitution functions would be useful. The inability to manipulate subscript lists is a problem of both data structures and primitives that lack.

The dyadic iota function has its arguments in the wrong order. Presently, the "universe" to be searched is the left argument, as is true of the characteristic function, ∈ . Furthermore, the universe is the argument most likely to be an expression. If arguments were interchanged, a pair of parentheses could often be eliminated and readability improved. For example, contrast `(STRING≠' ')⍳1` (present form) with `1⍳STRING≠' '` (proposed form). With this approach, the shape of the result is the same as the shape of the function's left argument, as is now true for ∈ .

No criticism of APL primitives would be complete without questioning the use of subscripts on functions and operators. Why should subscripts used here follow the index origin? The use of nonintegral subscripts in lamination is not only confusing and ugly, but also demolishes the notion that subscripts on primitives are somehow "indexing" something, and so should follow the rules for array indices. Finally, that reduction on a scalar argument works only if the subscript is elided and not at all if it is written explicitly is almost comical.

CONCLUSION

The casual reader of this paper may get the impression that APL is so fraught with problems as to be practically useless! To the contrary, though it has faults and failings, APL remains an unusually clean, consistent, and practical language.

We must ask, however, how much further can the development and extension of APL go? At some point, new features and extensions appear "grafted on". It becomes harder and harder to design elegant, natural extensions that harmonize with the existing structure. To date, most proposed generalizations of APL have raised at least as many new problems as they have purported to remedy.

Perhaps this effect reaffirms the elegance and solid good design of the base language. What sculptor would attempt to improve on Michelangelo's "David"? Thinking in terms of extensions and generalizations, we must pose a multitude of questions. To what extent do the decisions of the past, right or wrong, and the hypotheses upon which APL is built, constrain further development of the language?

The danger of not recognizing when to stop adding features to a language has already been demonstrated by the history of FORTRAN, BASIC, COBOL, and PL/I. To what extent are we willing (or unwilling) to change or even question early design decisions and assumptions in order to maintain compatibility? Is it worse to continue further extensions and adding new features and try to pretend that the result is still APL? When is APL not APL?

Maybe the wisest strategy is to leave well enough alone and stop tampering with the language. Recognizing APL's strengths and shortcomings, perhaps the next step is to begin anew from fundamentals to create a language that is to APL as APL is to FORTRAN.

Is there anyone who can perform the same leap of genius which gave rise to APL? Are we ready for this next step when it happens? Or has APL conservatism blinded us to new ideas?

POST SCRIPTUM

A lot of questions are raised in this paper. I hope readers will be encouraged to seek answers. APL could not have been created, nurtured, and brought to maturity without a certain singlemindedness of purpose on the part of many devoted people. It is, in large part, the transformation of this dedication of early workers into parochialism in others which prompted me to write this paper.

The opinions expressed here are wholly my own, and may not reflect the position of any other person or organization. I wish to thank the several friends and colleagues who have helped me to refine my thoughts and words to this stage. A special appreciation goes to Lawrence M. Breed for many fruitful discussions that contributed substantially to the form and the content of the work presented here.

REFERENCES

P. S. Abrams, An Interpreter for "Iverson Notation", Tech. Rept. CS47, Computer Science Dept., Stanford University (1966)

P. S. Abrams, An APL Machine, Ph.D. Dissertation, Stanford University (1970)

P. S. Abrams, "Program Writing, Rewriting, and Style", APL Congress 73 (1973) 1-8

J. A. Brown, A Generalization of APL, Ph.D. Dissertation, Syracuse University (1971)

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, London: Academic Press (1972)

E. W. Dijkstra, "The Humble Programmer", Communications of the ACM, 15, 10 (October 1972) 859-866

E. M. Edwards, "Generalized Arrays (Lists) in APL", APL Congress 73 (1973) 99-105

A. D. Falkoff and K. E. Iverson, APLSV User's Manual, Philadelphia: IBM (1973)

S. L. Gerhart, Verification of APL Programs, Ph.D. Dissertation, Carnegie-Mellon University (1972)

Z. Ghandour and J. E. Mezei, "General Arrays, Operators and Functions", IBM Journal of Research and Development, 17, 4 (July 1973) 335-352

L. Gilman and A. J. Rose, APL - An Interactive Approach, New York: John Wiley & Sons, Inc. (1974)

P. Gjerløv, H. J. Helms, and J. Nielsen, APL Congress 73, Amsterdam: North-Holland Publishing Co. (1973)

L. R. Harris, "A Logical Control Structure for APL", APL Congress 73 (1973) 203-210

K. E. Iverson, A Programming Language, New York: John Wiley & Sons, Inc. (1962)

K. E. Iverson, Elementary Functions: An Algorithmic Approach, Chicago: Science Research Associates, Inc. (1966)

R. A. Kelley, "APLGOL, An Experimental Structured Programming Language", IBM Journal of Research and Development, 17, 1 (January 1973) 69-73

R. H. Lathwell, and J. E. Mezei, "A Formal Description of APL", Colloque APL, Paris: IRIA (1971)

D. McCracken, "Whither APL", Datamation, 16, 11 (15 September 1970) 53-55

E. E. McDonnell, "Complex Floor", APL Congress 73, (1973) 299-305

T. More Jr., "Axioms and Theorems for a Theory of Arrays", IBM Journal of Research and Development, 17, 2 (March 1973) 135-175

R. C. Murray, "On Tree Structure Extensions to the APL Language", APL Congress 73, (1973) 333-338

T. H. Puckett, "Improved Security in APL Application Packages", Proceedings of the Sixth International APL Users' Conference, Coast Community College District (1974) 438-441

A. J. Rose, "More About Multi-Adic Functions", APL Quote Quad, 2, 6 (March 1971) 3-4

J. L. Ryan, "Secure Applications Within an APL Environment", APL Congress 73 (1973) 407-414