A. D. Falkoff
K. E. Iverson

The Design of APL

Abstract: This paper discusses the development of APL, emphasizing and illustrating the principles underlying its design. The principle of simplicity appears most strongly in the minimization of rules governing the behavior of APL objects, while the principle of practicality is served by the design process itself, which relies heavily on experimentation. The paper gives the rationale for many specific design choices, including the necessary adjuncts for system management.

Introduction

This paper attempts to identify the general principles that guided the development of APL and its computer realizations, and to show the role these principles played in the evolution of the language. The reader will be assumed to be familiar with the current definition of APL [1]. A brief chronology of the development of APL is presented in an appendix.

Different people claiming to follow the same broad principles may well arrive at radically different designs; an appreciation of the actual role of the principles in design can therefore be communicated only by illustrating their application in a variety of specific instances. It must be remembered, of course, that in the heat of battle principles are not applied as consciously or systematically as may appear in the telling. Some notion of the evolution of the ideas may be gained from consulting earlier discussions, particularly Refs. 2–4.

The actual operative principles guiding the design of any complex system must be few and broad. In the present instance we believe these principles to be simplicity and practicality. Simplicity enters in four guises: *uniformity* (rules are few and simple), *generality* (a small number of general functions provide as special cases a host of more specialized functions), *familiarity* (familiar symbols and usages are adopted whenever possible), and *brevity* (economy of expression is sought). Practicality is manifested in two respects: concern with actual application of the language, and concern with the practical limitations imposed by existing equipment.

We believe that the design of APL was also affected in important respects by a number of procedures and circumstances. Firstly, from its inception APL has been developed by *using* it in a succession of areas. This emphasis on application clearly favors practicality and simplicity. The treatment of many different areas fostered generalization; for example, the general inner product was developed in attempting to obtain the advantages of ordinary matrix algebra in the treatment of symbolic logic.

Secondly, the lack of any machine realization of the language during the first seven or eight years of its development allowed the designers the freedom to make radical changes, a freedom not normally enjoyed by designers who must observe the needs of a large working population dependent on the language for their daily computing needs. This circumstance was due more to the dearth of interest in the language than to foresight.

Thirdly, at every stage the design of the language was controlled by a small group of not more than five people. In particular, the men who designed (and coded) the implementation were part of the language design group, and all members of the design group were involved in broad decisions affecting the implementation. On the other hand, many ideas were received and accepted from people outside the design group, particularly from active users of some implementation of APL.

Finally, design decisions were made by Quaker consensus; controversial innovations were deferred until they could be revised or reevaluated so as to obtain unanimous agreement. Unanimity was not achieved without cost in time and effort, and many divergent paths were explored and assessed. For example, many different notations for the circular and hyperbolic functions were entertained over a period of more than a year before the present scheme was proposed, whereupon it was quickly adopted. As the language grows, more effort is needed to explore the ramifications of any major innovation. Moreover, greater care is needed in introducing new facilities, to avoid the possibility of later retraction that would inconvenience thousands of users. An example of the degree of preliminary exploration that may be involved is furnished by the depth and diversity of the investigations reported in the papers by Ghandour and Mezei [5] and by More [6].

The character set

The typography of a language to be entered at a simple keyboard is subject to two major practical restrictions: it must be linear, rather than two-dimensional, and it must be printable by a limited number of distinct symbols.

When one is not concerned with an immediate machine realization of a language, there is no strong reason to so limit the typography and for this reason the language may develop in a freer *publication form.* Before the design of a machine realization of APL, the restrictions appropriate to a keyboard form were not observed. In particular, different fonts were used to indicate the rank of a variable. In the keyboard form, such distinctions can be made, if desired, by adopting classes of names for certain classes of things.

The practical objective of linearizing the typography also led to increased uniformity and generality. It led to the present bracketed form of indexing, which removes the rank limitation on arrays imposed by use of superscripts and subscripts. It also led to the regularization of the form of dyadic functions such as $N\alpha J$ and $N\omega J$ (later eliminated from the language). Finally, it led to writing inner and outer products in the linear form +.× and ∘.× and eventually to the recognition of such expressions as instances of the use of *operators.*

The use of arrays and of operators greatly reduced the demand for distinct characters in APL, but the limitations imposed by the normal 88-symbol typewriter keyboard fostered two innovations which greatly increased the utility of the 88 symbols: the systematic use of most function symbols to represent both a dyadic and a monadic function, as suggested in conventional notation by the double use of the minus sign to represent both subtraction (a *dyadic* function) and negation (a *monadic* function); and the use of composite characters formed by typing one symbol over another (through the use of a backspace), as in ⌽ and ! and ⍟.

It was necessary to restrict the alphabetic characters to a single font and capitals were chosen for readability. Italics were initially favored because of their common use for denoting variables in mathematics, but were finally chosen primarily because they distinguished the letter *O* from the digit 0 and letters like *L* and *T* from the graphic symbols ⌊ and ⊤.

To allow the possibility of adding complete alphabetic fonts by overstriking, the underscore (_), diaeresis (¨), overbar (¯), and quad (⎕) were provided. In the APL\360 realization, only the underscore is used in this way. The inclusion of the overbar on the typeball fortunately filled a need we had not anticipated—a symbol for negative constants, distinct from the symbol for the negation function. The quad proved a useful symbol alone and in combination (as in ⍞), and the diaeresis still remains unassigned.

The SELECTRIC® typewriter imposed certain practical limitations on the placement of symbols on the keyboard, e.g., only narrow characters can appear in the upper row of the typing element. Within these limitations we attempted to make the keyboard easy to learn by grouping related symbols (such as the relations) in a rational order and by making mnemonic associations between letters and the functions associated with them in the shifted case (such as the *magnitude* function | with *M*, and the membership symbol ∊ with *E*).

Valence and order of execution

The *valence* of a function is the number of arguments it takes; APL primitives have valences of 1 (monadic functions) and 2 (dyadic functions), and user-defined functions may have a valence of 0 as well. The form for all APL primitives follows the familiar model of arithmetic, that is, the symbol for a dyadic function occurs between its arguments (as in 3+4) and the symbol for a monadic function occurs before its argument (as in -4).

A function f of valence greater than two is conventionally written in the form $f(a,b,c,d)$. This can be construed as a monadic function F applied to the vector argument a,b,c,d, and this interpretation is used in APL. In the APL\360 realization, the arguments $a,b,c,$ and d must share a common structure. The definition and implementation of generalized arrays, whose elements include *enclosed* arrays, will, of course, remove this restriction.

The result of any primitive APL function depends only on its immediate arguments, and the interpretation of each part of an APL statement is therefore localized. Likewise, the interpretation of each statement is independent of other statements in a program. This independence of context contributes significantly to the readability and ease of implementation of the language.

The order of execution of an APL expression is controlled by parentheses in the familiar way, and parentheses are used for no other purpose. The order is otherwise determined by one simple rule: the right argument of any function is the value of the entire expression following it. In particular, there is no precedence among functions; all functions, user-defined as well as primitive, are treated alike.

This simple rule has several consequences of practical advantage to the user:

a) An unparenthesized expression is easy to read from left to right because the first function encountered is the major function, the next is the major function in its right argument, etc.
b) An unparenthesized expression is also easy to read from right to left because this is the order in which it is executed.
c) If T is any vector of numerical terms, then the present rule makes the expressions `-/T` and `÷/T` very useful: the former is the alternating sum of T and the latter is the alternating product. Moreover, a continued fraction may be written without parentheses in the form `3+÷4+÷5+÷6`, and the efficient evaluation of a polynomial can be written without parentheses in the form `3+X×4+X×5+X×6`.

The rule that multiplication is executed before addition and that the power function is executed before multiplication has been long accepted in mathematics. In discarding any established rule it is wise to speculate on the reasons for its adoption and on whether they still apply. This rule makes parentheses unnecessary in the writing of polynomials, and this alone appears to be a sufficient reason for its original adoption. However, in APL a polynomial can be written more perspicuously in the form `+/C×X*E`, which also requires no parentheses. The question of the order of execution has been discussed in several places: Falkoff et al. [2,3], Berry [7], and Appendix A of Iverson [8].

The order in which isolated parts of a statement, such as the parts `(X+4)` and `(Y-2)` in the statement `(Y+4)×(Y-2)`, are executed is normally immaterial, but does matter when repeated specifications are permitted in a statement as in `(A←2)+A`. Although the use of such expressions is poor practice, it is desirable to make the interpretation unequivocal: the rule adopted (as given in Lathwell and Mezei [9]) is that the rightmost function or specification which can be performed is performed first.

It is interesting to note that the use of embedded assignment was first suggested during the course of the implementation when it was realized that special steps were needed to prevent it. The order of executing isolated parts of a statement was at first left unspecified (as stated in Falkoff and Iverson [1]) to allow freedom in implementation, since isolated parts could then be executed in parallel on any machine offering parallel processing. However, embedded assignment found such wide use that an unambiguous definition became essential to fix the behavior of programs moving from system to system.

Another aspect of the order of execution is the order among statements, which is normally taken as the order of appearance, except as modified by explicit *branches*. In the publication form of the language branches were denoted by arrows drawn from a branch point to the set of possible destinations, and the drawing of branch arrows is still to be recommended as an adjunct for clarifying the structure of a program (Iverson [10], page 3).

In formalizing branching it was necessary to introduce only one new concept (denoted by →) and three simple conventions: 1) continuing with the statement indicated by the first element of a vector argument of →, or with the next statement in sequence if the argument is an empty vector, 2) terminating the function if the indicated continuation is not the index of a statement in the program, and 3) the use of *labels*, local names defined by the indices of juxtaposed statements. At first labels were treated as local variables, but it was found to be more convenient in both use and implementation to treat them as local constants.

Since the branch arrow can be followed by any valid expression it provides convenient multi-way conditional branches. For example, if L is a Boolean vector and S is a corresponding set of statement numbers (often formed as the catenation of a set of labels), then `→L/S` provides a `(1+⍴L)`-way branch (to one of the elements of S or falling through if every element of L is zero); if I is an empty vector or an index to the vector S, then `→S[I]` provides a similar `(1+⍴L)`-way branch.

Programming languages commonly incorporate special forms of sequence control, typified by the DO statement of FORTRAN. These forms are excluded from APL because their cost in complication of the language outweighs their utility. The array operations in APL obviate many instances of iteration, and those which remain can be represented in a variety of ways. For example, grouping the initialization, modification, and testing of the control variable at the head of the iterated segment provides a particularly perspicuous arrangement. Moreover, specialized sequence control statements are usually context dependent and necessarily introduce new rules.

Conditional statements of the IF THEN ELSE type are not only context dependent, but their inherent limitation to a sequence of binary choices often leads to awkward constructions. These, and other, special sequence control forms can usually be modeled readily in APL and provided as application packages if desired.

Scalar functions

The emphasis on generality is illustrated in the definitions of many of the scalar functions. For example, the definition of the factorial is not limited to non-negative integers but is extended in the manner of the gamma function. Similarly, the residue is extended to all numbers in a simple and useful way: M|N is defined as the smallest (in magnitude) among the quantities N-M×I (where I is an integer) which lie in the range from 0 to M. If no such quantity exists (as in the case where M is zero) then the restriction to the range 0 to M is discarded, that is, 0|X is X. As another example, 0*0 is defined as 1 because that is the limiting value of X*Y when the point 0 0 is approached along any path other than the X axis, and because this definition is needed to make the common general form of writing a polynomial (in which the constant term C is written as C×X*0) applicable when the value of the argument X is zero.

The urge to generality must be tempered to avoid setting traps for the unwary, and compromise is sometimes necessary. For example, X÷0 could be defined as infinity (i.e., the largest representable number in an implementation) so as to obviate special treatment of the case Y=0 when computing the arc tangent of X÷Y, but is instead defined to yield a domain error. Nevertheless, 0÷0 is given the value 1, in spite of the fact that the mathematical argument for it is much weaker than that for 0*0, because it was deemed desirable to avoid an error stop in this case.

Eventually it will be desirable to be able to set separate limits on domains to suit various classes of users. For example, an implementation that incorporates complex numbers must yield a result for the expression ¯1*.5 but should admit of being set to yield a domain error for a user studying elementary arithmetic. The experienced user should be permitted to use an implementation in a mode that gives him complete control of domain and other errors, i.e., an error should not stop execution but should give necessary information about the error in a form which can be used by the program in which it occurs. Such a facility has not yet been incorporated in APL implementations.

A very general and useful set of functions was introduced by adopting the relation symbols < ≤ = ≥ > ≠ to represent functions (i.e., propositions) rather than assertions. The result of any proposition was defined to be 0 or 1 (rather than, say, *true* or *false*) so that it would lie in the domain of other arithmetic functions. Thus X=Y and X≠Y represent general comparisons, but if X and Y are integers then X=Y is the Kronecker delta and X≠Y is its inverse; if X and Y are Boolean variables, then X≠Y is the *exclusive-or* and X≤Y is material implication. This definition also allows expressions that incorporate both relational and arithmetic functions (such as (2=+/[1]0=S∘.|S)/S←⍳N, which yields the primes up to integer N). Moreover, identities among Boolean functions are more evident when expressed in these terms than when expressed in more conventional symbols.

The adoption of the relation symbols as functions does not preclude their use as *assertions* in informal sentences. For example, although one might feel compelled to substitute "X≤Y is true" for "X≤Y" in the sentence "If X≤Y then (X<Y)∨(X=Y)", there is no more reason to do so than to substitute "Bob is there is true" for "Bob is there" in the sentence which begins "If Bob is there then . . ."

Although we strove to adopt familiar symbols and usage, any clash with the principle of uniformity was invariably resolved in favor of uniformity. For example, familiar symbols (such as + - × ÷) are used where possible, but anomalies such as |X| for magnitude and *N!* for factorial are regularized to |X and !N. Notation such as X^N for power and $\binom{M}{N}$ for the binomial coefficient are replaced by regular dyadic forms X*N and M!N. Elision of the times sign is not permitted; this allows the use of multiple-character names and avoids confusion between multiplication, as in X(X+3), and the application of a function, as in F(X+3).

Moreover, each of the primitive scalar functions in APL is extended to arrays in exactly the same way. In particular, if V and W are vectors the expressions V×W and 3+V are permitted as well as the expressions V+W and 3×V, although only the latter pair would be permitted (in the sense used in APL) in conventional vector algebra.

One view of simplicity might exclude as redundant those functions which are easily expressed in terms of others. For example, ⌈X may be written as -⌊-X, and ⌈/X may be written as -⌊/-X, and ∧/L may be written as ~∨/~L. From another viewpoint it is simpler to use a more complete or symmetric set of primitives, since one need not remember which of a pair is provided and how to express the other in terms of it. In APL, completeness has been favored. For example, symbols are provided for all of the nontrivial logical functions although all are easily expressed in terms of a small subset of them.

The use of the circle to denote the whole family of functions related to the circular functions is a practical technique for conserving symbols as well as a useful generalization. It leads to many convenient expressions involving reduction and inner and outer products (such as 1 2 3∘.○X for a table of sines, cosines and tangents). Moreover, anyone wishing to use the symbol SIN for the sine function can define the function SIN as either 1○X (for radian arguments) or 1○X×180÷○1 (for degree arguments). The notational scheme employed for the circular functions must clearly be used with discretion; it could be used to replace all monadic functions by a single dyadic function with an integer left argument to encode each monadic function.

Operators

The dot in the expression M+.×N is an example of an *operator*; it takes functions (in this case + and ×) as arguments and produces a new function called an *inner product*. (In elementary mathematics the term *operator* is also used as a synonym for *function*, but in APL we eschew this usage.) The evolution of operators in APL furnishes an example of growing generality which has as yet been neither fully exploited nor fully regularized.

The operators now in APL were introduced one by one (reduction, then inner product, then outer product, then axis operators such as ⌽[*I*]) without being recognized as members of a class. When this class property was recognized it was apparent that the operators had not been given a consistent syntax and that the notation should eventually be regularized to give operators the same syntax as functions, i.e., an operator taking two arguments occurs between its (function) arguments (as in +.×) and an operator taking one argument appears in front of it. It also became evident that our treatment of operators had introduced a useful heirarchy into the order of execution, operators being executed before functions.

The recognition of operators as such has also made clear the much broader role they might be expected to play—derivative and integral operators are only two of many useful operators that must be added to the language.

The use of the outer product operator furnishes a clear example of a significant process in the evolution of the language: when a new facility is introduced it takes considerable time to recognize the many ways in which it can be used and therefore to appreciate its role in the further development of the language. The notation $\alpha^j(n)$ (later regularized to *N*⍺*J*) had been introduced early to represent a *prefix* vector, i.e., a Boolean vector of *N* elements with *J* leading 1's. Some thought had been given to extending the definition to a *vector J* (perhaps to yield an *N*-column matrix whose rows were prefix vectors determined by the elements of *J*) but no decision had been taken. When considering such an extension we normally communicate by defining any proposed notation in terms of existing primitives. After the outer product was introduced the proposed extension was written simply as *J*∘.≥⍳*N*, and it became clear that the function α was now redundant.

One should not conclude from this example that every function or set of functions easily expressed in terms of another is discarded as redundant; judgment must be exercised. In the present instance the ⍺ was discarded partly because it was too restrictive, i.e., the outer product form could be applied to yield a host of related functions (such as *J*∘.<⍳*N* and *J*∘.<⌽⍳*N*) not all of which were expressible in terms of the prefix and suffix functions ⍺ and ω. As mentioned in the discussion of scalar functions, the completeness of an obvious family of functions is also a factor to be considered.

Operators are attractive from several points of view. Because they provide a scheme for denoting whole classes of related functions, they offer uniformity of expression and great economy of symbols. The conciseness of expression that they allow can also be directly related to efficiency of implementation. Moreover, they introduce a new level of generality which plays an important role in the formal manipulability of the language.

Formal manipulation

APL is rich in identities and is therefore amenable to a great deal of fruitful formal manipulation. For example, many of the familiar identities of ordinary matrix algebra extend to inner products other than +.×, and de Morgan's law and other dualities extend to inner and outer products on arrays. The emphasis on generality, uniformity, and simplicity is likely to lead to a language rich in identities, but our emphasis on identities has been such that it should perhaps be enunciated as a separate and important guiding principle. Indeed, the preface to Iverson [10] cites one chapter (on the logical calculus) as illustration of "the formal manipulability of the language and its utility in theoretical work". A variety of identities is treated in [10] and [11], and a schema for proofs in APL is presented in [12].

Two examples will be used to illustrate the role of identities in the development of the language. The identity

```
(+/X)=(+/U/X)++/(~U)/X
```

applies for any numerical vector *X* and logical vector *U*. Maintaining this identity for the case where *U* is a vector of zeros forces one to define the sum over an empty vector as zero. A similar identity holds for reduction by any associative and commutative function and leads one to define the reduction of an empty array by any function as the identity element of that function.

The dyadic transpose *I*⍉*A* performs a general permutation on the coordinates of *A* as specified by the argument *I*. The monadic transpose is a special case which, in order to yield ordinary matrix transpose for an array of rank two, was initially defined to interchange the last two coordinates. It was later realized that the identity

```
∧/,(M+.×N)=⍉(⍉N)+.×⍉M
```

expected to hold for matrices would not hold for higher rank arrays. To make the identity true in general, the monadic transpose was defined to reverse the order of the coordinates as follows:

```
∧/,(⍉A)=(⌽⍳⍴⍴A)⍉A.
```

Moreover, the form chosen for the left argument of the dyadic transpose led to the following important identity:

```
∧/,(I⍉J⍉A)=I[J]⍉A.
```

Execute and format

In designing an executable language there is a fundamental choice to be made: Is the statement of an expression to be taken as an order to evaluate it, or must the evaluation be indicated by an explicit function in the language? This decision was made very early in the development of APL, albeit with little deliberation. Nevertheless, once the choice became manifest, early in the development of the implementation, it was applied uniformly in all situations.

There were some arguments against this, of course, particularly in the application of a function to its arguments, where it is often useful to be able to "call by name," which requires that the evaluation of the argument be deferred. But if implemented literally (i.e., if functions could be defined with this as an option) then names per se would have to be known to the language and would constitute an additional object type with its own rules of behavior and specialized primitive functions. A deliberate effort had been made to eliminate unnecessary type distinctions, as in the uniform language treatment of numbers regardless of their internal representation, and this point of view prevailed. In the interest of keeping the semantic rules simple, the idea of "call by name" was rejected as a primitive concept in APL.

Nevertheless, there are important cases where the formal argument of a function should not be evaluated at the time of invocation—as in the application of a generalized root finder to an arbitrary function. There are also situations where it is useful to inhibit evaluation of an expression, as in certain conditional forms, and the need for some treatment of the problem was clear. The basis for a solution was at hand in the form of character arrays, which were already objects of the language. Effectively, putting quotes around a statement inhibits its execution by making it a data item, a character array subject to the normal language functions. To get the effect of working with names, or with expressions to be conditionally evaluated, it was only necessary to introduce the notion of "unquote," or more properly "execute," as a function that would cause a character array to be evaluated as if it were the same expression without the inhibition.

The actual introduction of the execute function did not come for some time after its recognition as the likely solution. The development that preceded its final acceptance into APL illustrates several design principles.

The concept of an execute function is a very powerful one. In a sense, it makes the language "self-conscious," and introduces endless possibilities for obscurity in programs. This might have been a reason for not allowing it, but we had long since realized that a general-purpose language cannot be made foolproof and remain effective. Furthermore, APL is easily partitioned, and beginning users, or users of application packages, need not know about more sophisticated aspects of the language. The real issues were whether the function was of sufficiently broad utility, whether it could be defined simply, and whether it was perhaps a special case of a more general capability that should be implemented instead. There was also the need to establish a symbol for it.

The case for general utility was easily made. The execute function does allow names to be used as arguments to functions without the need for a new data type; it provides the means for generating variables under program control, which can be useful, for example, in managing data that do not conveniently fit into rectangular arrays; it allows the construction and execution of statements under program control; and in interpretive implementations it provides conversion from characters to numbers at machine speeds.

The behavior of the execute function is simply described: it treats a character array argument as a representation of an APL statement and attempts to evaluate or execute the statement so represented. System commands and attempts to enter function definition mode are not valid APL statements and are excluded from the domain of execute. It can be said that, except for these exclusions, execute acts upon a character array as if the elements of the array were entered at a terminal in the immediate execution mode.

Incidentally, there was pressure to arbitrarily include system commands in the domain of execute as a means of providing access to other workspaces under program control in order to facilitate work with large collections of data. This was resisted on the basis that the execute function should not allow by subterfuge what was otherwise disallowed. Indeed, consideration of this aspect of the behavior of execute led to the removal of certain anomalies in function definition and a clarification of the role of the escape characters) and ∇.

The question of generality has not been finally settled. Certainly, the execute function could be considered a member of a class that includes constructs like those of the lambda calculus. But it is not necessary to have the ultimate answer in order to proceed, and the simplicity of the definition adopted gives some assurance that generalizations are not being foreclosed.

For some time during its experimental implementation the symbol for execute was the epsilon. This was chosen for obvious mnemonic reasons and because no other monadic use was made of this symbol. As thought was being given to another new function—format—it was observed that over some part of each of their domains format and execute were inverses. Furthermore, over these parts of their domains they were strongly related to the functions encode and decode, and we therefore adopted their symbols overstruck by the symbol ∘.

The format function furnishes another example of a primitive whose behavior was first defined and long experimented with by means of APL defined functions. These defined functions were the *DFT* (Decimal Format) and *EFT* (Exponential Format) familiar to most users of the APL system. The main advantage of the primitive format function over these definitions is its much more efficient use of computer time.

The format function has both a dyadic and a monadic definition, but the execute function is monadic only. This leaves the way open for a related dyadic function, for which there has been no dearth of suggestions, but none will be adopted until more experience has been gained in the use of what we already have.

System commands and other environmental facilities

The definition of APL is purely abstract: the objects of the language, arrays of numbers and characters, are acted upon by the primitive functions in a manner independent of their representation and independent of any practical interpretation placed upon them. The advantages of such an abstract definition are that it makes the language truly machine independent, and avoids bias in favor of particular application areas. But not everything in a computing system is abstract, and provision must be made to manage system resources and otherwise communicate with the environment in which the language functions operate.

Maintaining the abstract nature of the language in a real computing system therefore seemed to imply a need for language-like facilities in some sense outside of APL. The need was first met by the use of system commands, which are syntactically not part of APL, and are also excluded from dynamic use within APL programs. They provided a simple and, in some ways, convenient answer to the problem of system management, but proved insufficient because the actions and information provided by them are often required dynamically.

The exclusion of system commands from programs was based more strongly on engineering considerations than on a theoretic compulsion, since the syntactic distinction alone sets them apart from the language, but there remained a reluctance to allow such syntactic anomalies in a program. The real issue, which was whether the functions provided by the system commands were properly the province of APL, was tabled for the time being, and defined functions that mimic the actions of certain of them were introduced to allow dynamic execution. The functions so provided were those affecting only the environment within a workspace, such as width and origin, while those that would have affected major physical resources of the system were still excluded for engineering reasons.

These environmental defined functions were based on the use of still another class of functions—called "I-beams" because of the shape of the symbol used for them—which provide a more general facility for communication between APL programs and the less abstract parts of the system. The I-beam functions were first introduced by the system programmers to allow them to execute System/360 instructions from within APL programs, and thus use APL as a direct aid in their programming activity. The obvious convenience of functions of this kind, which appeared to be part of the language, led to the introduction of the monadic I-beam function for direct use by anyone. Various arguments to this function yielded information about the environment such as available space and time of day.

Though clearly an ad hoc facility, the I-beam functions appear to be part of the language because they obey APL syntax and can be executed from within an APL program. They were too useful to do without in the absence of a more rational solution to the problem, and so were graced with the designation "system-dependent functions," while we continued to use the system and think about the general problem of communication among the subsystems composing it.

Shared variables

The logical basis for a generalized communication facility in APL\360 was laid in 1964 with the publication of the formal description of System/360 [2]. It was then observed that the interaction between concurrent "asynchronous" processes (programs) could be completely comprehended by an interface comprising variables that were shared by the cooperating processes. (Another facility was also used, where one program forced a branch in another, but this can be regarded as a derivative representation based on variables shared between one program and a processor that drives the other.) It was not until six or seven years later, however, that the full force of this observation was brought to bear on the practical problem of controlling in an organic way the environment in which APL programs run.

Three processors can be identified during the execution of an APL program: APL, or the processor that actually executes the program; the *system*, or host that manages libraries and other environmental factors, which in APL\360 is the System/360 processor; and the user, who may be observing and processing output or providing input to the program. The link between APL and system is the set of I-beam functions, that between user and system is the set of system commands, and between user and APL, the quad and quote-quad. With the exception of the quote-quad, which is a true variable, all these links are constructs on the interfaces rather than the interfaces themselves.

It can be seen that the quote-quad is shared by the user and APL. Characteristically, a value assigned to it in a program is presented to the user at the terminal, who utilizes this information as he sees fit. If later read by the program, the value of the quote-quad then has no fixed relationship to what was earlier specified by the program. The values written and read by the program are *a fortiori* APL objects—abstract arrays—but they may have practical significance to the user-processor, suggesting, for example, that an experimental observation be made and the results entered at the keyboard.

Using the quote-quad as the paradigm for their behavior, a general facility for shared variables was designed and implemented starting in late 1969 (see Lathwell [13]). The underlying concept was to provide communication across the boundary between independent processors by explicitly establishing certain variables as being shared between them. A shared variable is syntactically indistinguishable from others and may be used normally either on the right or left of an assignment arrow.

Although motivated most strongly at the time by a need to provide a "file and I/O" capability for APL\360, the shared variable facility satisfied other needs as well, a significant criterion for the inclusion of a new feature in the language. It provides for general communication, not only between APL and the host system, but also between APL programs running concurrently at different terminals, which is in a sense a more fundamental use of the idea.

Perhaps as important as the practical use of the facility is the potency that an implementation lends to the concept of shared variables as a basis for understanding communication in any system. With respect to APL\360, for example, we had long used the term "distinguished variable" in discussing the interface between APL and system, meaning thereby variables, like trace and stop vectors, which hold control or state information. It is now clear that "distinguished variables" are shared variables, distinguished from ordinary variables by the fact of their being shared, and further qualified by their membership in a particular interface. In principle, the environment and resources of APL\360 could be completely controlled through the use of an appropriate set of such distinguished variables.

System functions

In a given application area it is usually easier to work with APL augmented by defined functions, designed to embody the significant concepts of the area, than with the primitive functions of the language alone. Such defined functions, together with the relevant variables or data objects, constitute an application language, or application extension. Managing the resources or environment of an APL computing system is a particular application, in which the data objects are the distinguished variables that define the interface between APL and system.

For convenience, the defined functions constituting an application extension for system management should behave differently from other defined functions, at least to the extent of being available at all times, like the primitives, without having to be copied from workspace to workspace. Such ubiquity requires that the names of these functions be distinguished from those a user might invent. This distinction can only be made, if APL is to remain essentially context independent, by the establishment of a class of reserved names. This class has been defined as names starting with the quad character, and functions having such names are called *system functions*. A similar naming convention applies to distinguished variables, or *system variables*, as they are now called.

In principle, system functions work with system variables that are independently identifiable. In practice, the system variables in a particular situation may not be available explicitly, and the system functions may be locked. This can come about because direct access to the interface by the user is deemed undesirable for technical reasons, or because of economic considerations such as efficiency or protection of proprietary rights. In such situations system functions are superficially distinguishable from primitive functions only by virtue of the naming convention.

The present I-beam functions behave like system functions. Fortunately, there are only two of them: the monadic function that is familiar to all users of APL, and the dyadic function that is still known mostly to system programmers. Despite their usefulness, these functions are hardly to be taken as examples of good application language design, depending as they do on arbitrary numerical arguments to give them meaning, and having no meaningful relationships with each other. The monadic I-beams are more like read-only variables—changeable constants, as it were—than functions. Indeed, except for their syntax, they behave precisely like shared variables where the processor on the other side replaces the value between each reference on the APL side.

The shared variable facility itself requires communication between APL and system in order to establish a desired interface between APL and cooperating processors. The prospect of inventing new system commands for this, or otherwise providing an ad hoc facility, was most distasteful, and consideration of this problem was a major factor in leading toward the system function concept. It was taken as an indication of the validity of the shared variable approach to communication when the solution to the problem it engendered was found within the conceptual framework it provided, and this solution also proved to be a basis for clarifying the role of facilities already present.

In due course a set of system functions must be designed to parallel the facilities now provided by system commands and go beyond them. Aside from the obvious advantage of being dynamically executable, such a set of system functions will have other advantages and some disadvantages. The major operational advantage is that the system functions will be able to use the full power of APL to generate their arguments and exploit their results. Countering this, there is the fact that this power has a price: the automatic name isolation provided by the extralingual system commands will not be available to the system functions. Names used as arguments will have to be presented as character arrays, which is not a disadvantage in programs, although it is less convenient for casual keyboard entry than is the use of unadorned names in system commands.

A more profound advantage of system functions over system commands lies in the possibility of designing the former to work together constructively. System commands are foreclosed from this by the rudimentary nature of their syntax; they do constitute a language, but one having no constructive potential.

Workspaces, files, and input-output

The workspace organization of APL\360 libraries serves to group together functions and variables intended to work together, and to render them active or inactive as a group, preserving the state of the computation during periods of inactivity. Workspaces also implicitly qualify the names of objects within them, so that the same name may be used independently in a multiplicity of workspaces in a given system. These are useful attributes; the grouping feature, for example, contributes strongly to the convenience of using APL by obviating the linkage problems found in other library systems.

On the other hand, engineering decisions made early in the development of APL\360 determined that the workspaces be of fixed size. This limits the size of objects that can be managed within them and often becomes an inconvenience. Consequently, as usage of APL\360 developed, a demand arose for a "file" facility, at first to work with large volumes of data under program control, and later to utilize data generated by other systems. There was also a demand to make use of high-speed input and output equipment. As noted in an earlier section, these demands led in time to the development of the shared variable facility. Three considerations were paramount in arriving at this solution.

One consideration was the determination to maintain the abstract nature of APL. In particular, the use of primitive functions whose definitions depend on the representation of their arguments was to be avoided. This alone was sufficient to rule out the notion of a file as a formal concept in the language. APL has primitive array structures that either encompass the logical structure of files or can be extended to do so by relatively simple functions defined on them. The user of APL may regard any array or collection of arrays as a file, and in principle should be able to use the data so organized without regard to the medium on which these arrays may be stored.

The second consideration was the not uncommon observation that files are used in two ways, as a medium for exchange of information and as a dynamic extension of working storage during computation (see Falkoff [14]). In keeping with the principle just noted, the proper solution to the second problem must ultimately be the removal of workspace size limitations, and this will probably be achieved in the course of general developments in the industry. We saw no prospect of a satisfactory direct solution being achieved locally in a reasonable time, so attention was concentrated on the first problem in the expectation that, with a good general communication facility, on-line storage devices could be used for workspace extension at least as effectively as they are so used in other systems.

The third consideration was one of generality. One possible approach to the communication problem would have been to increase the roster of system commands and make them dynamically executable, or add variations to the I-beam functions to manage specific storage media and I/O equipment or access methods. But in addition to being unpleasant because of its ad hoc nature, this approach did not promise to be general enough. In working interactively with large collections of data, for example, the possible functional variations are almost limitless. Various classes of users may be allowed access for different purposes under a variety of controls, and unless it is intended to impose restrictive constraints ahead of time, it is futile to try to anticipate the solutions to particular problems. Thus, to provide a communication facility by accretion appeared to be an endless task.

The shared variable approach is general enough because, by making the interface explicitly available with primitive controls on the behavior of the shared variable, it provides only the basic communication mechanism. It then remains for the specific problem to be managed by bringing to bear on it the full power of APL on one side, and that of the host system on the other. The only remaining question is one of performance: does the shared variable concept provide the basis for an effective implementation? This question has been answered affirmatively as a result of direct experimentation.

The net effect of this approach has been to provide for APL an application extension comprising the few system functions necessary to manage shared variables. Actual file or I/O applications are managed, as required, by user-defined functions. The system functions are used only to establish sharing, and the shared variables are then used for the actual transfer of information between APL workspaces and file or I/O processors.

Appendix. Chronology of APL development

The development of APL was begun in 1957 as a necessary tool for writing clearly about various topics of interest in data processing. The early development is described in the preface of Iverson [10] and Brooks and Iverson [15]. Falkoff became interested in the work shortly after Iverson joined IBM in 1960, and used the language in his work on parallel search memories [16]. In early 1963 Falkoff began work on a formal description of System/360 in APL and was later joined in this work by Iverson and Sussenguth [2].

Throughout this early period the language was used by both Falkoff and Iverson in the teaching of various topics at various universities and at the IBM Systems Research Institute. Early in 1964 Iverson began using it in a course in elementary functions at the Fox Lane High School in Bedford, New York, and in 1966 published a text that grew out of this work [8]. John L. Lawrence (who, as editor of the *IBM Systems Journal*, procured and assisted in the publication of the formal description of System/360) became interested in the use of APL at high school and college level and invited the authors to consult with him in the development of curriculum material based on the use of computers. This work led to the preparation of curriculum material in a number of areas and to the publication of an APL\360 Reference Manual by Sandra Pakin [17].

Although our work through 1964 had been focused on the language as a tool for communication among *people*, we never doubted that the same characteristics which make the language good for this purpose would make it good for communication with a machine. In 1963 Herbert Hellerman implemented a portion of the language on an IBM/1620 as reported in [18]. Hellerman's system was used by students in the high school course with encouraging results. This, together with our earlier work in education, heightened our interest in a full-scale implementation.

When the work on the formal description of System/360 was finished in 1964 we turned our attention to the problem of implementation. This work was brought to rapid fruition in 1965 when Lawrence M. Breed joined the project and, together with Philip S. Abrams, produced an implementation on the 7090 by the end of 1965. Influenced by Hellerman's interest in time-sharing we had already developed an APL typing element for the IBM 1050 computer terminal. This was used in early 1966 when Breed adapted the 7090 system to an experimental time-sharing system developed under Andrew Kinslow, allowing us the first use of APL in the manner familiar today. By November 1966, the system had been reprogrammed for System/360 and APL service has been available within IBM since that date. The system became available outside IBM in 1968.

A paper by Falkoff and Iverson [3] provided the first published description of the APL\360 system, and a companion paper by Breed and Lathwell [19] treated the implementation. R. H. Lathwell joined the design group in 1966 and has since been concerned primarily with the implementations of APL and with the use of APL itself in the design process. In 1971 he published, together with Jorge Mezei, a formal definition of APL in APL [9].

The APL\360 System benefited from the contributions of many outside of the central design group. The preface to the User's Manual [1] acknowledges many of these contributions.

References

1. A. D. Falkoff and K. E. Iverson, *APL\360 User's Manual*, IBM Corporation, (GH20-0683-1) 1970.
2. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal*, **3**, 198 (1964).
3. A. D. Falkoff and K. E. Iverson, "The APL\360 Terminal System", *Symposium on Interactive Systems for Experimental Applied Mathematics*, eds., M. Klerer and J. Reinfelds, Academic Press, New York, 1968.
4. A. D. Falkoff, "Criteria for a System Design Language," *Report on NATO Science Committee Conference on Software Engineering Techniques*, April 1970.
5. Z. Ghandour and J. Mezei, "General Arrays, Operators and Functions," *IBM J. Res. Develop.* **17**, 335 (1973, this issue).
6. T. More, "Axioms and Theorems for a Theory of Arrays—Part I," *IBM J. Res. Develop.* **17**, 135 (1973).
7. P. C. Berry, *APL\360 Primer*, IBM Corporation, (GH-20-0689-2) 1971.
8. K. E. Iverson, *Elementary Functions: An Algorithmic Treatment*, Science Research Associates, Chicago, 1966.
9. R. H. Lathwell and J. E. Mezei, "A Formal Description of APL," *Colloque APL*, Institut de Recherche d'-Informatique et d'Automatique, Rocquencourt, France, 1971.
10. K. E. Iverson, *A Programming Language*, Wiley, New York, 1962.
11. K. E. Iverson, "Formalism in Programming Languages," *Communications of the ACM*, **7**, 80 (February, 1964).
12. K. E. Iverson, *Algebra: an algorithmic treatment*, Addison-Wesley Publishing Co., Reading, Mass., 1972.
13. R. H. Lathwell, "System Formulation and APL Shared Variables," *IBM J. Res. Develop.* **17**, 353 (1973, this issue).
14. A. D. Falkoff, "A Survey of Experimental APL File and I/O Systems in IBM", *Colloque APL*, Institut de Recherche d'Informatique et D'Automatique, Rocquencourt, France, 1971.
15. F. P. Brooks and K. E. Iverson, *Automatic Data Processing*, Wiley, New York, 1963.
16. A. D. Falkoff, "Algorithms for Parallel Search Memories," *Journal of the ACM*, 488 (1962).
17. S., Pakin, *APL\360 Reference Manual*, Science Research Associates, Inc., Chicago, 1968.

18. H. Hellerman, "Experimental Personalized Array Translator System," *Communications of the ACM* **7**, 433 (July, 1964).
19. L. M. Breed and R. H. Lathwell, "Implementation of APL/360," *Symposium on Interactive Systems for Experimental Applied Mathematics*, eds., M. Klerer and J. Reinfelds, Academic Press, New York, 1968.

Received May 16, 1972

The authors are located at the IBM Data Processing Division Scientific Center, 3401 Market Street, Philadelphia, Pennsylvania 19104.