THE EVOLUTION OF APL

Adin D. Falkoff
Kenneth E. Iverson

Research Division
IBM Corporation

This paper is a discussion of the evolution of the APL language, and it treats implementations and applications only to the extent that they appear to have exercised a major influence on that evolution. Other sources of historical information are cited in References 1-3; in particular, The Design of APL [1] provides supplementary detail on the reasons behind many of the design decisions made in the development of the language. Readers requiring background on the current definition of the language should consult APL Language [4].

Although we have attempted to confirm our recollections by reference to written documents and to the memories of our colleagues, this remains a personal view which the reader should perhaps supplement by consulting the references provided. In particular, much information about individual contributions will be found in the Appendix to The Design of APL [1], and in the Acknowledgements in A Programming Language [10] and in APL\360 User's Manual [23]. Because Reference 23 may no longer be readily available, the acknowledgements from it are reprinted in Appendix A.

McDonnell's recent paper on the development of the notation for the circular functions [5] shows that the detailed evolution of any one facet of the language can be both interesting and illuminating. Too much detail in the present paper would, however, tend to obscure the main points, and we have therefore limited ourselves to one such example. We can only hope that other contributors will publish their views on the detailed developments of other facets of the language, and on the development of various applications of it.

The development of the language was first begun by Iverson as a tool for describing and analyzing various topics in data processing, for use in teaching classes, and in writing a book, Automatic Data Processing [6], undertaken together with Frederick P. Brooks, Jr., then a graduate student at Harvard. Because the work began as incidental to other work, it is difficult to pinpoint the beginning, but it was probably early 1956; the first explicit use of the language to provide communication between the designers and programmers of a complex system occurred during a leave from Harvard spent with the management consulting firm of McKinsey and Company in 1957. Even after others were drawn into the development of the language, this development remained largely incidental to the work in which it was used. For example, Falkoff was first attracted to it (shortly after Iverson joined IBM in 1960) by its use as a tool in his work in parallel search memories [7], and in 1964 we began to plan an implementation of the language to enhance its utility as a design tool, work which came to fruition when we were joined by Lawrence M. Breed in 1965.

The most important influences in the early phase appear to be Iverson's background in mathematics, his thesis work in the machine solutions of linear differential equations [8] for an economic input-output model proposed by Professor Wassily Leontief (who, with Professor Howard Aiken, served as thesis adviser), and Professor Aiken's interest in the newly-developing field of commercial applications of computers. Falkoff brought to the work a background in engineering and technical development, with experience in a number of disciplines, which had left him convinced of the overriding importance of simplicity, particularly in a field as subject to complication as data processing.

Although the evolution has been continuous, it will be helpful to distinguish four phases according to the major use or preoccupation of the period: academic use (to 1960), machine description (1961-1963), implementation (1964-1968), and systems (after 1968).

1. ACADEMIC USE

The machine programming required in Iverson's thesis work was directed at the development of a set of subroutines designed to permit convenient experimentation with a variety of mathematical methods. This implementation experience led to an emphasis on implementable language constructs, and to an understanding of the role of the representation of data.

The mathematical background shows itself in a variety of ways, notably:

1. In the use of functions with explicit arguments and explicit results; even the relations ($< \leq = \geq > \neq$) are treated as such functions.

2. In the use of logical functions and logical variables. For example, the compression function (denoted by /) uses as one argument a logical vector which is, in effect, the characteristic vector of the subset selected by compression.

3. In the use of concepts and terminology from tensor analysis, as in inner product and outer product and in the use of rank for the "dimensionality" of an array, and in the treatment of a scalar as an array of rank zero.

4. In the emphasis on generality. For example, the generalizations of summation (by $F/$), of inner product (by $F.G$), and of outer product (by $\circ.F$) extended the utility of these functions far beyond their original area of application.

5. In the emphasis on identities (already evident in [9]) which makes the language more useful for analytic purposes, and which leads to a uniform treatment of special cases as, for example, the definition of the reduction of an empty vector, first given in A Programming Language [10].

In 1954 Harvard University published an announcement [11] of a new graduate program in Automatic Data Processing organized by Professor Aiken. (The program was also reported in a conference on computer education [12]). Iverson was one of the new faculty appointed to prosecute the program; working under the guidance of Professor Aiken in the development of new courses provided a stimulus to his interest in developing notation, and the diversity of interests embraced by the program promoted a broad view of applications.

The state of the language at the end of the academic period is best represented by the presentation in A Programming Language [10], submitted for publication in early 1961. The evolution in the latter part of the period is best seen by comparing references 9 and 10. This comparison shows that reduction and inner and outer product were all introduced in that period, although not then recognized as a class later called operators. It also shows that specification was originally (in Reference 9) denoted by placing the specified name at the right, as in $P+Q \rightarrow Z$. The arguments (due in part to F.P. Brooks, Jr.) which led to the present form ($Z \leftarrow P+Q$) were that it better conformed to the mathematical form $Z=P+Q$, and that in reading a program, any backward reference to determine how a given variable was specified would be facilitated if the specified variables were aligned at the left margin. What this comparison does not show is the removal of a number of special comparison functions (such as the comparison of a vector with each row of a matrix) which were seen to be unnecessary when the power of the inner product began to be appreciated, as in the expression $M\wedge.=V$. This removal provides one example of the simplification of the language produced by generalizations.

2. MACHINE DESCRIPTION

The machine description phase was marked by the complete or partial description of a number of computer systems. The first use of the language to describe a complete computing system was begun in early 1962 when Falkoff discussed with Dr. W.C. Carter his work in the standardization of the instruction set for the machines that were to become the IBM System/360 family. Falkoff agreed to undertake a formal description of the machine language, largely as a vehicle for demonstrating how parallel processes could be rigorously represented. He was later joined in this work by Iverson when he returned from a short leave at Harvard, and still later by E.H. Sussenguth. This work was published as "A Formal Description of System/360" [13].

This phase was also marked by a consolidation and regularization of many aspects which had little to do with machine description. For example, the cumbersome definition of maximum and minimum (denoted in Reference 10 by $U\lceil V$ and $U\lfloor V$ and equivalent to what would now be written as $\lceil/U/V$ and $\lfloor/U/V$) was replaced, at the suggestion of Herbert Hellerman, by the present simple scalar functions. This simplification was deemed practical because of our increased understanding of the potential of reduction and inner and outer product.

The best picture of the evolution in this period is given by a comparison of A Programming Language [10] on the one hand, and "A Formal Description of System/360" [13] and "Formalism in Programming

Languages" [14] on the other. Using explicit page references to Reference 10, we will now give some further examples of regularization during this period:

1. The elimination of embracing symbols (such as $|X|$ for absolute value, $\lfloor X \rfloor$ for floor, and $\lceil X \rceil$ for ceiling) and replacement by the leading symbol only, thus unifying the syntax for monadic functions.

2. The conscious use of a single function symbol to represent both a <u>monadic</u> and a <u>dyadic</u> function (still referred to in Reference 10 as <u>unary</u> and <u>binary</u>).

3. The adoption of multi-character names which, because of the failure (page 11) to insist on no elision of the times sign, had been permitted (page 10) only with a special indicator.

4. The rigorous adoption of a right-to-left order of execution which, although stated (page 8) had been violated by the unconscious application of the familiar precedence rules of mathematics. Reasons for this choice are presented in <u>Elementary Functions</u> [15], in Berry's <u>APL\360 Primer</u> [16], and in <u>The Design of APL</u> [1].

5. The concomitant definition of reduction based on a right-to-left order of execution as opposed to the opposite convention defined on page 16.

6. Elimination of the requirement for parentheses surrounding an expression involving a relation (page 11). An example of the use without parentheses occurs near the bottom of page 241 of Reference 13.

7. The elimination of implicit specification of a variable (that is, the specification of some function of it, as in the expression $\iota S \leftarrow 2$ on page 81), and its replacement by an explicit inverse function (τ in the cited example).

Perhaps the most important developments of this period were in the use of a collection of concurrent autonomous programs to describe a system, and the formalization of <u>shared variables</u> as the means of communication among the programs. Again, comparisons may be made between the system of programs of Reference 13, and the more informal use of concurrent programs introduced on page 88 of Reference 10.

It is interesting to note that the need for a random function (denoted by the question mark) was first felt in describing the operation of the computer itself. The architects of the IBM System/360 wished to leave to the discretion of the designers of the individual machines of the 360 family the decision as to what was to be found in certain registers after the occurrence of certain errors, and this was done by stating that the result was to be random. Recognizing more general use for the function than the generation of random logical vectors, we subsequently defined the monadic question mark function as a scalar function whose argument specified the population from which the random elements were to be chosen.

3. IMPLEMENTATION

In 1964 a number of factors conspired to turn our attention seriously to the problem of implementation. One was the fact that the language was by now sufficiently well-defined to give us some confidence in its suitability for implementation. The second was the interest of Mr. John L. Lawrence who, after managing the publication of our description of System/360, asked for our consultation in utilizing the language as a tool in his new responsibility (with Science Research Associates) for developing the use of computers in education. We quickly agreed with Mr. Lawrence on the necessity for a machine implementation in this work. The third was the interest of our then manager, Dr. Herbert Hellerman, who, after initiating some implementation work which did not see completion, himself undertook an implementation of an array-based language which he reported in the <u>Communications of the ACM</u> [17]. Although this work was limited in certain important respects, it did prove useful as a teaching tool and tended to confirm the feasibility of implementation.

Our first step was to define a character set for APL. Influenced by Dr. Hellerman's interest in time-sharing systems, we decided to base the design on an 88-character set for the IBM 1050 terminal, which utilized the easily-interchanged Selectric® typing element. The design of this character-set exercised a surprising degree of influence on the development of the language.

As a practical matter it was clear that we would have to accept a linearization of the language (with no superscripts or subscripts) as well as a strict limit on the size of the primary character set. Although we expected these limitations to have a deleterious effect, and at first found unpleasant some of the linearity forced upon us, we now feel that the changes were beneficial, and that many led to important generalizations. For example:

1. On linearizing indexing we realized that the sub- and super-script form had inhibited the

use of arrays of rank greater than 2, and had also inhibited the use of several levels of indexing; both inhibitions were relieved by the linear form *A*[*I*;*J*;*K*].

2. The linearization of the inner and outer product notation (from $M\overset{+}{\times}N$ and $M\overset{\circ}{\times}N$ to *M*+.×*N* and *M*∘.×*N*) led eventually to the recognition of the operator (which was now represented by an explicit symbol, the period) as a separate and important component of the language.

3. Linearization led to a regularization of many functions of two arguments (such as *N*α*J* for $\alpha^J(n)$ and *A***B* for a^b) and to the redefinition of certain functions of two or three arguments so as to eliminate one of the arguments. For example, $\iota^J(n)$ was replaced by ι*N*, with the simple expression *J*+ι*N* replacing the original definition. Moreover, the simple form ι*N* led to the recognition that *J*≥ι*N* could replace *N*α*J* (for *J* a scalar) and that *J*∘.≥ι*N* could generalize *N*α*J* in a useful manner; as a result the functions α and ω were eventually withdrawn.

4. The limitation of the character set led to a more systematic exploitation of the notion of ambiguous valence, the representation of both a monadic and a dyadic function by the same symbol.

5. The limitation of the character set led to the replacement of the two functions for the number of rows and the number of columns of an array, by the single function (denoted by ρ) which gave the dimension vector of the array. This provided the necessary extension to arrays of arbitrary rank, and led to the simple expression ρρ*A* for the rank of *A*. The resulting notion of the dimension vector also led to the definition of the dyadic reshape function *D*ρ*X*.

6. The limitation to 88 primary characters led to the important notion of composite characters formed by striking one of the basic characters over another. This scheme has provided a supply of easily-read and easily-written symbols which were needed as the language developed further. For example, the quad, overbar, and circle were included not for specific purposes but because they could be used to overstrike many characters. The overbar by itself also proved valuable for the representation of negative numbers, and the circle proved convenient in carrying out the idea, proposed by E.E. McDonnell, of representing the entire family of (monadic) circular functions by a single dyadic function.

7. The use of multiple fonts had to be re-examined, and this led to the realization that certain functions were defined not in terms of the value of the argument alone, but also in terms of the form of the name of the argument. Such dependence on the forms of names was removed.

We did, however, include characters which could print above and below alphabetics to provide for possible font distinctions. The original typing element included both the present flat underscore, and a saw-tooth one (the praltriller as shown, for example, in Webster's Second), and a hyphen. In practice, we found the two underscores somewhat difficult to distinguish, and the hyphen very difficult to distinguish from the minus, from which it differed only in length. We therefore made the rather costly change of two characters, substituting the present delta and del (inverted delta) for the pralltriller and the hyphen.

In the placement of the character set on the keyboard we were subject to a number of constraints imposed by the two forms of the IBM 2741 terminal (which differed in the encoding from keyboard-position to element-position), but were able to devise a grouping of symbols which most users find easy to learn. One pleasant surprise has been the discovery that numbers of people who do not use APL have adopted the type element for use in mathematical typing. The first publication of the character set appears to be in Elementary Functions [15].

Implementation led to a new class of questions, including the formal definition of functions, the localization and scope of names, and the use of tolerances in comparisons and in printing output. It also led to systems questions concerning the environment and its management, including the matter of libraries and certain parameters such as index origin, printing precision, and printing width.

Two early decisions set the tone of the implementation work: 1) The implementation was to be experimental, with primary emphasis on flexibility to permit experimentation with language concepts, and with questions of execution efficiency subordinated, and 2) The language was to be compromised as little as possible by machine considerations.

These considerations led Breed and P.S. Abrams (both of whom had been attracted to our work by Reference 13) to propose and build an interpretive implementation in the summer of 1965. This was a batch system with punched card input, using a multi-character encoding of the primitive function symbols. It ran on the IBM 7090 machine and we were later able to experiment with it interactively, using the typeball previously designed, by placing the interpreter under an experimental time sharing monitor (TSM) available on a machine in a nearby IBM facility.

TSM was available to us for only a very short time, and in early 1966 we began to consider an implementation on System/360, work that started in earnest in July and culminated in a running system in the fall. The fact that this interpretive and experimental implementation also proved to be remarkably practical and efficient is a tribute to the skill of the implementers, recognized in 1973 by the award to the principals (L.M. Breed, R.H. Lathwell, and R.D. Moore) of ACM's Grace Murray Hopper Award. The fact that the many APL implementations continue to be largely interpretive may be attributed to the array character of the language which makes possible efficient interpretive execution.

We chose to treat the occurrence of a statement as an order to evaluate it, and rejected the notion of an explicit function to indicate evaluation. In order to avoid the introduction of "names" as a distinct object class, we also rejected the notion of "call by name". The constraints imposed by this decision were eventually removed in a simple and general way by the introduction of the execute function, which served to execute its character string argument as an APL expression. The evolution of these notions is discussed at length in the section on "Execute and Format" in The Design of APL [1].

In earlier discussions with a number of colleagues, the introduction of declarations into the language was urged upon us as a requisite for implementation. We resisted this on the general basis of simplicity, but also on the basis that information in declarations would be redundant, or perhaps conflicting, in a language in which arrays are primitive. The choice of an interpretive implementation made the exclusion of declarations feasible, and this, coupled with the determination to minimize the influence of machine considerations such as the internal representations of numbers on the design of the language, led to an early decision to exclude them.

In providing a mechanism by which a user could define a new function, we wished to provide six forms in all: functions with 0, 1, or 2 explicit arguments, and functions with 0 or 1 explicit results. This led to the adoption of a header for the function definition which was, in effect, a paradigm for the way in which a function was used. For example, a function F of two arguments having an explicit result would typically be used in an expression such as $Z \leftarrow A\ F\ B$, and this was the form used for the header.

The names for arguments and results in the header were of course made local to the function definition, but at the outset no thought was given to the localization of other names. Fortunately, the design of the interpreter made it relatively easy to localize the names by adding them to the header (separated by semicolons), and this was soon done. Names so localized were strictly local to the defined function, and their scope did not extend to any other functions used within it. It was not until the spring of 1968 when Breed returned from a talk by Professor Alan Perlis on what he called "dynamic localization" that the present scheme was adopted, in which name scopes extend to functions called within a function.

We recognized that the finite limits on the representation of numbers imposed by an implementation would raise problems which might require some compromise in the definition of the language, and we tried to keep these compromises to a minimum. For example, it was clear that we would have to provide both integer and floating point representations of numbers and, because we anticipated use of the system in logical design, we wished to provide an efficient (one bit per element) representation of logical arrays as well. However, at the cost of considerable effort and some loss of efficiency, both well worthwhile, the transitions between representations were made to be imperceptible to the user, except for secondary effects such as storage requirements.

Problems such as overflow (i.e., a result outside the range of the representations available) were treated as domain errors, the term domain being understood as the domain of the machine function provided, rather than as the domain of the abstract mathematical function on which it was based.

One difficulty we had not anticipated was the provision of sensible results for the comparison of quantities represented to a limited precision. For example, if X and Y were specified by $Y \leftarrow 2 \div 3$ and $X \leftarrow 3 \times Y$, then we wished to have the comparison $2 = X$ yield 1 (representing true) even though the representation of the quantity X would differ slightly from 2.

This was solved by introducing a comparison tolerance (christened fuzz by L.M. Breed, who knew of its use in the Bell Interpreter [18]) which was multiplied by the larger in magnitude of the arguments to give a tolerance to be applied in the comparison. This tolerance was at first fixed (at $1E^{-}13$) and was later made specifiable by the user. The matter has proven more difficult than we first expected, and discussion of it still continues [19, 20].

A related, but less serious, question was what to do with the rational root of a negative number, a question which arose because the exponent (as in the expression ¯8*2÷3) would normally be presented as an approximation to a rational. Since we wished to make the mathematics behave "as you thought it did in high school" we wished to treat such cases properly at least for rationals with denominators of reasonable size. This was achieved by determining the result sign by a continued fraction expansion of the right argument (but only for negative left arguments) and worked for all denominators up to 80 and "most" above.

Most of the mathematical functions required were provided by programs taken from the work of the late Hirondo Kuki in the FORTRAN IV Subroutine Library. Certain functions (such as the inverse hyperbolics) were, however, not available and were developed, during the summers of 1967 and 1968, by K. M. Brown, then on the faculty of Cornell University.

The fundamental decision concerning the systems environment was the adoption of the concept of a workspace. As defined in "The APL\360 Terminal System" [21]:

> APL\360 is built around the idea of a workspace, analogous to a notebook, in which one keeps work in progress. The workspace holds both defined functions and variables (data), and it may be stored into and retrieved from a library holding many such workspaces. When retrieved from a library by an appropriate command from a terminal, a copy of the stored workspace becomes active at that terminal, and the functions defined in it, together with all the APL primitives, become available to the user.
>
> The three commands required for managing a library are "save", "load", and "drop", which respectively store a copy of an active workspace into a library, make a copy of a stored workspace active, and destroy the library copy of a workspace. Each user of the system has a private library into which only he can store. However, he may load a workspace from any of a number of common libraries, or if he is privy to the necessary information, from another user's private library. Functions or variables in different workspaces can be combined, either item by item or all at once, by a fourth command, called "copy". By means of three cataloging commands, a user may get the names of workspaces in his own or a common library, or get a listing of functions or variables in his active workspace.

The language used to control the system functions of loading and storing workspaces was not APL, but comprised a set of _system commands_. The first character of each system command is a right parenthesis, which cannot occur at the left of a valid APL expression, and therefore acts as an "escape character", freeing the syntax of what follows. System commands were used for other aspects such as sign-on and sign-off, messages to other users, and for the setting and sensing of various system parameters such as the index origin, the printing precision, the print width, and the _random link_ used in generating the pseudo-random sequence for the random function.

When it first became necessary to name the implementation we chose the acronym formed from the book title _A Programming Language_ [10] and, to allow a clear distinction between the language and any particular implementation of it, initiated the use of the machine name as part of the name of the implementation (as in APL\1130 and APL\360). Within the design group we had until that time simply referred to "the language".

A brief working manual of the APL\360 system was first published in November 1966 [22], and a full manual appeared in 1968 [23]. The initial implementation (in FORTRAN on an IBM 7090) was discussed by Abrams [24], and the time-shared implementation on System/360 was discussed by Breed and Lathwell [25].

3. SYSTEMS

Use of the APL system by others in IBM began long before it had been completed to the point described in _APL\360 User's Manual_ [23]. We quickly learned the difficulties associated with changing the specifications of a system already in use, and the impact of changes on established users and programs. As a result we learned to appreciate the importance of the relatively long period of development of the language which preceded the implementation; early implementation of languages tends to stifle radical change, limiting further development to the addition of features and frills.

On the other hand, we also learned the advantages of a running model of the language in exposing anomalies and, in particular, the advantage of input from a large population of users concerned with a broad range of applications. This use quickly exposed the major deficiencies of the system.

Some of these deficiencies were rectified by the generalization of certain functions and the addition of others in a process of gradual evolution. Examples include the extension of the catenation function to apply to arrays other than vectors and to permit *lamination*, and the addition of a generalized matrix inverse function discussed by M.A. Jenkins [26].

Other deficiencies were of a systems nature, concerning the need to communicate between concurrent APL programs (as in our description of System/360), to communicate with the APL system itself within APL rather than by the ad hoc device of system commands, to communicate with alien systems and devices (as in the use of file devices), and the need to define functions within the language in terms of their representation by APL arrays. These matters required more fundamental innovations and led to what we have called the *system* phase.

The most pressing practical need for the application of APL systems to commercial data processing was the provision of file facilities. One of the first commercial systems to provide this was the File Subsystem reported by Sharp [27] in 1970, and defined in a SHARE presentation by L.M. Breed [28], and in a manual published by Scientific Time Sharing Corporation [29]. As its name implies, it was not an integral part of the language but was, like the system commands, a practical ad hoc solution to a pressing problem.

In 1970 R.H. Lathwell proposed what was to become the basis of a general solution to many systems problems of APL\360, a *shared variable processor* [30] which implemented the shared variable scheme of communication among processors. This work culminated in the APLSV System [31] which became generally available in 1973.

Falkoff's "Some Implications of Shared Variables" [32] presents the essential notion of the shared variable system as follows:

> A user of early APL systems essentially had what appeared to be an "APL machine" at his disposal, but one which lacked access to the rest of the world. In more recent systems, such as APLSV and others, this isolation is overcome and communication with other users and the host system is provided for by *shared variables*.
>
> Two classes of shared variables are available in these systems. First, there is a *general shared variable* facility with which a user may establish arbitrary, *temporary*, interfaces with other users or with auxiliary processors. Through the latter, communication may be had with other elements of the host system, such as its file subsystem, or with other systems altogether. Second, there is a set of *system variables* which define parts of the *permanent* interface between an APL program and the underlying processor. These are used for interrogating and controlling the computing environment, such as the origin for array indexing or the action to be taken upon the occurrence of certain exceptional conditions.

4. A DETAILED EXAMPLE

At the risk of placing undue emphasis on one facet of the language, we will now examine in detail the evolution of the treatment of numeric constants, in order to illustrate how substantial changes were commonly arrived at by a sequence of small steps.

Any numeric constant, including a constant vector, can be written as an expression involving APL primitive functions applied to decimal numbers as, for example, in `3.14×10*-5` and `-2.718` and `(3.14×10*-5),(-2.718),5`. At the outset we permitted only non-negative decimal constants of the form `2.718`, and all other values had to be expressed as compound statements.

Use of the monadic negation function in producing negative values in vectors was particularly cumbersome, as in `(-4),3,(-5),-7`. We soon realized that the adoption of a specific "negative" symbol would solve the problem, and familiarity with Beberman's work [33] led us to the adoption of his "high minus" which we had, rather fortuitously, included in our character set. The constant vector used above could now be written as `¯4,3,¯5,¯7`.

Solution of the problem of negative numbers emphasized the remaining awkwardness of factors of the form `10*N`. At a meeting of the principals in Chicago, which included Donald Mitchell and Peter Calingaert of Science Research Associates, it was realized that the introduction of a scaled form of constant in the manner used in FORTRAN would not complicate the syntax, and this was soon adopted.

These refinements left one function in the writing of any vector constant, namely, catenation. The straightforward execution of an expression for a constant vector of N elements involved N-1 catenations of scalars with vectors of increasing length, the handling of roughly `.5×N×N+1` elements in all. To avoid gross inefficiencies in the input of a constant vector from the keyboard, catenation was therefore given special treatment in the original implementation.

This system had been in use for perhaps six months when it occurred to Falkoff that since commas were not required in the normal representation of a matrix, vector constants might do without them as well. This seemed outrageously simple, and we looked for flaws. Finding none we adopted and implemented the idea immediately, but it took some time to overcome the habit of writing expressions such as (3,3)ρ*X* instead of 3 3ρ*X*.

5. CONCLUSIONS

Nearly all programming languages are rooted in mathematical notation, employing such fundamental notions as functions, variables, and the decimal (or other radix) representation of numbers, and a view of programming languages as part of the longer-range development of mathematical notation can serve to illuminate their development.

Before the advent of the general-purpose computer, mathematical notation had, in a long and painful evolution well-described in Cajori's history of mathematical notations [34], embraced a number of important notions:

1. The notion of assigning an alphabetic name to a variable or unknown quantity (Cajori, Secs. 339-341).

2. The notion of a _function_ which applies to an _argument or arguments_ to produce an _explicit_ result which can itself serve as argument to another function, and the associated adoption of specific symbols (such as + and ×) to denote the more common functions (Cajori, Secs. 200-233).

3. _Aggregation_ or _grouping_ symbols (such as the parentheses) which make possible the use of composite expressions with an unambiguous specification of the order in which the component functions are to be executed (Cajori, Secs. 342-355).

4. Simple, uniform representations for numeric quantities (Cajori, Secs. 276-289).

5. The treatment of quantities without concern for the particular representation used.

6. The notion of treating vectors, matrices, and higher-dimensional arrays as entities, which had by this time become fairly widespread in mathematics, physics, and engineering.

With the first computer languages (machine languages) all of these notions were, for good practical reasons, dropped; variable names were represented by "register numbers", application of a function (as in *A*+*B*) was necessarily broken into a sequence of operations (such as "Load register 801 into the Addend register, Load register 802 into the Augend register, etc."), grouping of operations was therefore non-existent, the various functions provided were represented by numbers rather than by familiar mathematical symbols, results depended sharply on the particular representation used in the machine, and the use of arrays, as such, disappeared.

Some of these limitations were soon removed in early "automatic programming" languages, and languages such as FORTRAN introduced a limited treatment of arrays, but many of the original limitations remain. For example, in FORTRAN and related languages the size of an array is not a language concept, the asterisk is used instead of any of the familiar mathematical symbols for multiplication, the power function is represented by two occurrences of this symbol rather than by a distinct symbol, and concern with representation still survives in declarations.

APL has, in its development, remained much closer to mathematical notation, retaining (or selecting one of) established symbols where possible, and employing mathematical terminology. Principles of simplicity and uniformity have, however, been given precedence, and these have led to certain departures from conventional mathematical notation as, for example, the adoption of a single form (analogous to 3+4) for dyadic functions, a single form (analogous to -4) for monadic functions, and the adoption of a uniform rule for the application of all scalar functions to arrays. This relationship to mathematical notation has been discussed in _The Design of APL_ [1] and in "Algebra as a Language" which occurs as Appendix A in _Algebra: an algorithmic treatment_ [35].

The close ties with mathematical notation are evident in such things as the reduction operator (a generalization of sigma notation), the inner product (a generalization of matrix product), and the outer product (a generalization of the outer product used in tensor analysis). In other aspects the relation to mathematical notation is closer than might appear. For example, the order of execution of the conventional expression *F G H* (*X*) can be expressed by saying that the right argument of each function is the value of the entire expression to its right; this rule, extended to dyadic as well as monadic functions, is the rule used in APL. Moreover, the term _operator_ is used in the same sense as in "derivative operator" or "convolution operator" in mathematics, and to avoid conflict it is not used as a synonym for function.

As a corollary we may remark that the other major programming languages, although known to the designers of APL, exerted little or no influence, because of their radical departures from the line of development of mathematical notation which APL continued. A concise view of the current use of the language, together with comments on matters such as writing style, may be found in Falkoff's review of the 1975 and 1976 International APL Congresses [36].

Although this is not the place to discuss the future, it should be remarked that the evolution of APL is far from finished. In particular, there remain large areas of mathematics, such as set theory and vector calculus, which can clearly be incorporated in APL through the introduction of further operators.

There are also a number of important features which are already in the abstract language, in the sense that their incorporation requires little or no new definition, but are as yet absent from most implementations. Examples include complex numbers, the possibility of defining functions of ambiguous valence (already incorporated in at least two systems [37, 38]), the use of user defined functions in conjunction with operators, and the use of selection functions other than indexing to the left of the assignment arrow.

We conclude with some general comments, taken from The Design of APL [1], on principles which guided, and circumstances which shaped, the evolution of APL:

> 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.

ACKNOWLEDGEMENTS

For critical comments arising from their reading of this paper, we are indebted to a number of our colleagues who were there when it happened, particularly P.S. Abrams of Scientific Time Sharing Corporation, R.H. Lathwell and R.D. Moore of I.P. Sharp Associates, and L.M. Breed and E.E. McDonnell of IBM Corporation.

REFERENCES

1. Falkoff, A.D., and K.E. Iverson, The Design of APL, IBM Journal of Research and Development, Vol.17, No.4, July 1973, pages 324-334.
2. The Story of APL, Computing Report in Science and Engineering, IBM Corp., Vol.6, No.2, April 1970, pages 14-18.
3. Origin of APL, a videotape prepared by John Clark for the Fourth APL Conference , 1974, with the participation of P.S. Abrams, L.M. Breed, A.D. Falkoff, K.E. Iverson, and R.D. Moore. Available from Orange Coast Community College, Costa Mesa, California.
4. Falkoff, A.D., and K.E. Iverson, APL Language, Form No. GC26-3847, IBM Corp., 1975
5. McDonnell, E. E., The Story of o, APL Quote-Quad, Vol. 8, No. 2, ACM, SIGPLAN Technical Committee on APL (STAPL), December, 1977, pages 48-54.
6. Brooks, F.P., and K.E. Iverson, Automatic Data Processing, John Wiley and Sons, 1973.
7. Falkoff, A.D., Algorithms for Parallel Search Memories, Journal of the ACM, Vol. 9, 1962, pages 488-511.
8. Iverson, K.E., Machine Solutions of Linear Differential Equations: Applications to a Dynamic Economic Model, Harvard University, 1954 (Ph.D. Thesis).
9. Iverson, K.E., The Description of Finite Sequential Processes, Proceedings of the Fourth London Symposium on Information Theory, Colin Cherry, Editor, 1960, pages 447-457.
10. Iverson, K.E., A Programming Language, John Wiley and Sons, 1962.
11. Graduate Program in Automatic Data Processing, Harvard University, 1954, (Brochure).
12. Iverson, K.E., Graduate Research and Instruction, Proceedings of First Conference on Training Personnel for the Computing Machine Field, Wayne State University, Detroit, Michigan, June, 1954, Arvid W. Jacobson, Editor, pages 25-29.
13. Falkoff, A.D., K.E. Iverson, and E.H. Sussenguth, A Formal Description of System/360, IBM Systems Journal, Vol 4, No. 4, October 1964, pages 198-262.
14. Iverson, K.E., Formalism in Programming Languages, Communications of the ACM, Vol.7, No.2, February 1964, pages 80-88.
15. Iverson, K.E., Elementary Functions, Science Research Associates, 1966.
16. Berry, P.C., APL\360 Primer, IBM Corporation (GH20-0689), 1969.
17. Hellerman, H., Experimental Personalized Array Translator System, Communications of the ACM, Vol.7, No.7, July 1964, pages 433-438.
18. Wolontis, V.M., A Complete Floating Point Decimal Interpretive System, Technical Newsletter No. 11, IBM Applied Science Division, 1956.
19. Lathwell, R.H., APL Comparison Tolerance, APL 76 Conference Proceedings, Association for Computing Machinery, 1976, pages 255-258.
20. Breed, L.M., Definitions for Fuzzy Floor and Ceiling, Technical Report No. TR03.024, IBM Corporation, March 1977.
21. Falkoff, A.D., 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, pages 22-37.
22. Falkoff, A.D., and K.E. Iverson, APL\360, IBM Corporation, November 1966.
23. Falkoff, A.D., and K.E. Iverson, APL\360 User's Manual, IBM Corporation, August 1968.
24. Abrams, P.S., An Interpreter for Iverson Notation, Technical Report CS47, Computer Science Department, Stanford University, 1966.
25. Breed, L.M., and R.H. Lathwell, The Implementation of APL\360, Symposium on Interactive Systems for Experimental and Applied Mathematics, eds. M. Klerer and J. Reinfelds, Academic Press, New York, 1968, pages 390-399.

26. Jenkins, M.A., The Solution of Linear Systems of Equations and Linear Least Squares Problems in APL, IBM Technical Report No. 320-2989, 1970.

27. Sharp, Ian P., The Future of APL to benefit from a new file system, Canadian Data Systems, March 1970.

28. Breed, L.M., The APL PLUS File System, Proceedings of SHARE XXXV, August, 1970, page 392.

29. APL PLUS File Subsystem Instruction Manual, Scientific Time Sharing Corporation, Washington, D.C., 1970.

30. Lathwell, R.H., System Formulation and APL Shared Variables, IBM Journal of Research and Development, Vol.17, No.4, July 1973, pages 353-359.

31. Falkoff, A.D., and K.E. Iverson, APLSV User's Manual, IBM Corporation, 1973.

32. Falkoff, A.D., Some Implications of Shared Variables, Formal Languages and Programming, R. Aguilar, ed., North Holland Publishing Company, 1976, pages 65-75. Reprinted in APL 76 Conference Proceedings, Association for Computing Machinery, pages 141-148.

33. Beberman, M., and H.E. Vaughn, High School Mathematics Course 1, Heath, 1964.

34. Cajori, F., A History of Mathematical Notations, Vol. I, Notations in Elementary Mathematics, The Open Court Publishing Co., La Salle, Illinois, 1928.

35. Iverson, K.E., Algebra: an algorithmic treatment, Addison Wesley, 1972.

36. Falkoff, A.D., APL75 and APL76: an overview of the Proceedings of the Pisa and Ottawa Congresses, ACM Computing Reviews, Vol. 18, No. 4, April, 1977, Pages 139-141.

37. Weidmann, Clark, APLUM Reference Manual, University of Massachusetts, Amherst, Massachusetts, 1975.

38. Sharp APL Technical Note No. 25, I.P. Sharp Associates, Toronto, Canada.

APPENDIX A

Reprinted from APL\360 User's Manual [23]

ACKNOWLEDGEMENTS

The APL language was first defined by K.E. Iverson in A Programming Language (Wiley, 1962) and has since been developed in collaboration with A.D. Falkoff. The APL\360 Terminal System was designed with the additional collaboration of L.M. Breed, who with R.D. Moore*, also designed the S/360 implementation. The system was programmed for S/360 by Breed, Moore, and R.H. Lathwell, with continuing assistance from L.J. Woodrum⊛, and contributions by C.H. Brenner, H.A. Driscoll**, and S.E. Krueger**. The present implementation also benefitted from experience with an earlier version, designed and programmed for the IBM 7090 by Breed and P.S. Abrams⊛⊛.

The development of the system has also profited from ideas contributed by many other users and colleagues, notably E.E. McDonnell, who suggested the notation for the signum and the circular functions.

In the preparation of the present manual, the authors are indebted to L.M. Breed for many discussions and suggestions; to R.H. Lathwell, E.E. McDonnell, and J.G. Arnold*⊛ for critical reading of successive drafts; and to Mrs. G.K. Sedlmayer and Miss Valerie Gilbert for superior clerical assistance.

A special acknowledgement is due to John L. Lawrence, who provided important support and encouragement during the early development of APL implementation, and who pioneered the application of APL in computer-related instruction.

*I.P. Sharp Associates, Toronto, Canada.

⊛General Systems Architecture, IBM Corporation, Poughkeepsie, N.Y.

**Science Research Associates, Chicago, Illinois.

⊛⊛Computer Science Department, Stanford University, Stanford, California.

*⊛Industry Development, IBM Corporation, White Plains, N.Y.