A Personal View of APL

—by K. E. Iverson
Toronto, Ontario

This essay portrays a personal view of the development of several influential dialects of APL: APL2 and J. The discussion traces the evolution of the treatment of arrays, functions, and operators, as well as function definition, grammar, terminology, and spelling.

IT IS NOW 35 YEARS since Professor Howard Aiken instituted a computer science program at Harvard, a program that he called Automatic Data Processing. It is almost that long since I began to develop, for use in writing and teaching in that program, the programming language that has come to be known as APL.

Although I have consulted original papers and compared my recollections with those of colleagues, this remains a personal essay that traces the development of my own thinking about notation. In particular, my citation of the work of others does not imply that they agree with my present interpretation of their contributions. In speaking of design decisions I use the word we to refer to the small group associated with the early implementation, a group that included Adin Falkoff, Larry Breed, and Dick Lathwell, and is identified in "The Design of APL"[1] and "The Evolution of APL."[2] These papers contain full treatments of various aspects of the development of APL that are given scant attention here.

Because my formal education was in mathematics, the fundamental notions in APL have been drawn largely from mathematics. In particular, the notions of arrays, functions, and operators were adopted at the outset, as illustrated by the following excerpt from *A Programming Language.*[3]

An operation (such as summation) which is applied to all components of a vector is called reduction.... Thus, +/*x* is the sum, ×/*x* is the product, and ∨/*x* is the logical sum of the components of a vector *x*.

The phrase +/*x* alone illustrates the three aspects: a *function* +, an operator / (so named from the term used by Heaviside[4] for an entity that applies to a function to produce a related derived function), and an array *x*.

The present discussion is organized by topic, tracing the evolution of the treatments of arrays, functions, and operators; as well as that of other matters such as function definition, grammar, terminology, and spelling (that is, the representation of primitives).

As stated at the outset, the initial motive for developing APL was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects.

When I retired from paid employment, I turned my attention back to this matter and soon concluded that the essential tool required was a dialect of APL that:

- Is available as "shareware," and is inexpensive enough to be acquired by students as well as by schools
- Can be printed on standard printers
- Runs on a wide variety of computers
- Provides the simplicity and generality of the latest thinking in APL

The result has been J, first reported in Reference 5.

Work began in the summer of 1989 when I first discussed my desires with Arthur Whitney. He proposed the use of C for implementation, and produced (on one page and in one afternoon) a working fragment that provided only one function (+), one operator (/), one-letter names, and arrays limited to ranks 0 and 1, but did provide for boxed arrays and for the use of the copula for assigning names to any entity.

I showed this fragment to others in the hope of interesting someone competent in both C and APL to take up the work, and soon recruited Roger Hui, who was attracted in part by the unusual style of C programming used by Arthur, a style that made heavy use of preprocessing facilities to permit writing further C in a distinctly APL style.

Roger and I then began a collaboration on the design and implementation of a dialect of APL (later named J by Roger), first deciding to roughly follow "A Dictionary of APL"[6] and to impose no requirement of compatibility with any existing dialect. We were assisted by suggestions from many sources, particularly in the design of the spelling scheme (E. B. Iverson and A. T. Whitney) and in the treatment of cells, items, and formatting (A. T. Whitney, based on his work on SHARP/HP[7] and on the dialect A reported at the APL89 conference in New York).

E. E. McDonnell of Reuters provided C programs for the mathematical functions (which apply to complex numbers as well as to real), D. L. Orth of IBM ported the system to the IBM RISC System/ 6000* in time for the APL90 conference, and L. J. Dickey of the University of Waterloo provided assistance in porting the system to a number of other computers.

The features of J that distinguish it from most other APL dialects include:

1. A spelling scheme that uses ASCII characters in one- or two-letter words
2. Convenient international use, provided by facilities for alternative spellings for the *national use* characters of ASCII, and by facilities to produce the error messages in any desired language
3. Emphasis on *major cells* or items; for example, reduction (`f/`) applies `f` between items, and application of f between cells of lesser rank is obtained by using the rank operator
4. The function argument to scan (`\`) is, like all functions, ambivalent. Scan applies the monadic case of the function rather than the dyadic. Thus, the traditional sum scan is given by `+/\ a` rather than by `+\a`, and `<\a` boxes the partitions provided by the scan.
5. A number of other partitioning adverbs are provided, including suffix scan (`\.`), windows of width k (as in `k f\a`), and *oblique* (`/.`).
6. Use of the hook and fork (discussed later) and various new operators together with the use of the copula to assign names to functions. These facilities permit the extensive use of tacit programming in which the arguments of a function are not explicitly referred to in its definition, a form of programming that requires no reparsing of the function on execution, and therefore provides some of the efficiency of compilation. (See Reference 8.)
7. An immediate and highly readable display of the definition of a function `f` obtained by simply entering `f`

Significant use of J in teaching will, of course, require the development of textual material using it. Three steps have been, taken toward this goal:

1. The dictionary of J includes 45 frames of tutorial material (suitable for slides) that are brief treatments in J of topics from a dozen different areas.
2. At the urging of L. B. Moore of I. P. Sharp Associates, I prepared for distribution at APL89 a booklet called *Tangible Math,* designed for independent study of elementary mathematics. It was based on the use of Sharp[9] shareware for the IBM PC, and required no reference to an APL manual. I have since produced a J version of *Tangible Math.*[10]
3. At a four-hour hands-on workshop for teachers of mathematics organized by Anthony Camacho of I-APL[11] and funded by the British APL Association, Anthony and I used *Tangible Math to* expose the participants to the advantages of executable mathematical notation. The teachers left with a copy of J and with enough experience to continue the use of J on their own. Such workshops could be used to bring teachers to a point where they could develop their own treatments of isolated topics, and eventually of complete subjects, on their own.

In the three decades of APL development, many different ideas have been proposed and explored, and many have been abandoned. Those that survived have done so through incorporation in one or more implementations that define the many dialects of APL.

These dialects fall into several families, two of which have been particularly influential. I refer to them by the names of their most recent exemplars—APL2[12] on the one hand, and J on the other—and sketch the development of these families in a later section.

In the remainder of the essay I largely confine my remarks to those dialects that have influenced, and been influenced by, my own thinking. This emphasis is intended not to denigrate the dialects not mentioned, but to keep the discussion focused and to leave their exposition to others more conversant with them.

Although my motive for producing a new dialect was for use in teaching, this dialect has led to much greater emphasis on a style of programming called *functional* by Backus,[13] and defined in J as tacit programming (because arguments are not referred to explicitly). These matters are addressed in the section on tacit programming.

Terminology

Although terminology was not among the matters given serious attention at the outset, it will be helpful to adopt some of the later terminology immediately. Because of our common mathematical background, we initially chose mathematical terms. For example, the sentence

```
b←(+\a)−.×a←2 3 5 7
```

illustrates certain parts of speech, for which we adopted the mathematical terms shown on the left as follows:

Functions or operators	`+ × −`	Verbs
Constant (vector)	`2 3 5 7`	Noun (list)
Variables	*a b*	Pronouns
Operator	`\`	Adverb
Operator	`.`	Conjunction
	`( )`	Punctuation
	`←`	Copula

I now prefer terms drawn from natural language, as illustrated by the terms shown on the right. Not only are they familiar to a broader audience, but they clarify the purposes of the parts of speech and of certain relations among them:

1. A verb specifies an "action" upon a noun or nouns.
2. An adverb applies to a verb to produce a related verb; thus +\ is the verb "partial sums."
3. A conjunction applies to two verbs, in the manner of the copulative conjunction *and* in the phrase "run and hide."
4. A name such as a or b behaves like a pronoun, serving as a surrogate for any referent linked to it by a copula. The mathematical term *variable* applied to a name x in the identity (x+1)×(x+3) equals x^2+4x+3 serves to emphasize that the relation holds for any value of x, but the term is often inappropriate for pronouns used in programming.
5. Although numeric lists and tables are commonly used to represent the vectors and matrices of mathematics, the terms list and *table* are much broader and simpler, and suggest the essential notions better than do the mathematical terms.
6. To avoid ambiguity due to the two uses of the term operator in mathematics (for both a function and a Heaviside operator) I usually use only the terms *adverb* and *conjunction,* but continue to use either *function* or *verb, list* or *vector,* and *table* or *matrix,* as seems appropriate.

Spelling

In natural languages the many words used are commonly represented (or *spelled)* in an *alphabet* of a small number of characters. In programming languages the words or primitives of the languages (such as *sin* and = :) are commonly represented by an expanded alphabet that includes a number of graphic symbols such as + and =.

When we came to implement APL, the alphabet then widely available on computers was extremely limited, and we decided to exploit a feature of our company's newly-developed Selectric* typewriter, whose changeable typing element allowed us to design our own alphabet of 88 characters. By limiting the English alphabet to one case (majuscules), and by using the backspace key to *produce composite* characters, we were able to design a spelling scheme that used only one-character words for primitives.

Moreover, the spelling scheme was quite mnemonic in an international sense, relying on the appearance of the symbols rather than on names of the functions in any national language. Thus the phrase k↑x takes k elements from x, and ↓ denotes drop.

Because the use of the APL alphabet was relatively limited, it was not included in the standard ASCII alphabet now widely adopted. As a consequence, it was not available on most printers, and the printing and publication of APL material became onerous. Nevertheless, in spite of some experiments with "reserved words" in the manner of other programming languages, the original APL alphabet has remained the standard for APL systems.

The set of graphics in ASCII is much richer than the meager set available when the APL alphabet was designed, and it can be used in spelling schemes for APL primitives that still avoid the adoption of reserved words. Such a scheme using variable-length words was presented in Reference 6, and received limited use for communicating APL programs using standard printers, but was never adopted in any commercial implementation. A much simpler scheme using words of one or two letters was adopted in J, in a manner that largely retains, and sometimes enhances, the international mnemonic character of APL words.

In a natural language such as English, the process of word formation is clearly distinguished from parsing. In particular, word formation is static, the rhematic rules applying to an entire text quite independently of the meanings or grammatical classes of the words produced. Parsing, on the other hand, is dynamic, and proceeds according to the grammatical classes of phrases as they evolve. This is reflected in the use of such terms as *noun phrase* and *verb phrase.*

In programming languages this distinction is commonly blurred by combining word formation and parsing in a single process characterized as "syntax." In J, the word formation and parsing are distinct. In its implementations, each process is *tabledriven;* the parsing table being presented explicitly in the dictionary of J, and the rhematic rules being discussed only informally.

It is interesting to note that the words of early APL included "composite characters" represented by two elements of the underlying alphabet; these were mechanically superposed, whereas in J they appear side-by-side.

Functions

Functions were first adopted in the forms found in elementary mathematics, having one argument (as in $|b|$ and $-b$) or two (as in $a+b$ and $a-b$). In particular, each had an explicit result, so that functions could be articulated to form sentences, as in $|a-b|\div(a+b)$.

In mathematics, the symbol − is used to denote both the *dyadic* function *subtraction* (as in $a-b$) and the *monadic* function *negation* (as in $-b$). This ambivalent use of symbols was exploited systematically (as in ÷ for both *division* and reciprocal, and ⋆ for both power and *exponential) to* provide mnemonic links between related functions, and to economize on symbols.

The same motivations led us to adopt E. E. McDonnell's proposal to treat the monadic trigonometric (or circular) functions and related *hyperbolic* and *pythagorean* functions as a single family of dyadic functions, denoted by a circle. Thus *sine y and cosine y are* denoted by 1○y and 2○y, the numeric left argument being chosen so that its parity (even or odd) agrees with the parity of the function denoted, and so that a negative integer denotes the function inverse to that denoted by the corresponding positive integer. This scheme was a matter of following (with rather less justification) the important mathematical notion of treating the monadic functions *square, cube, square root,* etc. as special cases of the single dyadic power function.

When the language was formalized and linearized in APL\360,[14] anomalies such as x^y for power, xy for product, $|y|$ for magnitude, and M^i_j for indexing were replaced by `x*y` and `x×y` and `|y` and `M[i;j]`. At the same time, function definition was formalized, using *headers* of the form `Z←X F Y` and `Z←F Y` to indicate the definition of a dyadic or a monadic function. This form of header permitted the definition of functions having no explicit result (as in `X F Y`), and so-called *niladic* functions (as in `Z←F` and `F`) having no explicit arguments. These forms were adopted for their supposed convenience, but this adoption introduced functions whose articulation in sentences was limited.

In most later dialects such niladic and resultless functions were also adopted as primitives. In J they have been debarred completely, to avoid the problem of articulation, to avoid complications in the application of adverbs and conjunctions to them, and to avoid the following problem with the copula: if `g` is a niladic function that yields the noun `n`, and if `f←g`, then is `f` a niladic function equivalent to `g`, or is it the noun `n`?

In conventional mathematical notation, an expression such as $f(x,y,z)$ can be interpreted either as a function of three arguments, or as a function of one argument, that is, of the vector formed by the catenation of x, y, and z. Therefore the limitation of APL functions to at most two formal arguments does not limit the number of scalar arguments to which a function may apply.

Difficulties with nonscalar arguments first arose in indexing, and the forms such as `A[I;J;K]` and `A[I;;K]` that were adopted to deal with it introduced a "nonlocality" into the language: a phrase within brackets had to be treated as a whole rather than as the application of a sequence of functions whose results could each be assigned a name or otherwise treated as a normal result. Moreover, an index expression for an array `A` could not be written without knowing the rank of `A`.

The introduction of a function to produce an *atomic representation* of a noun (known as *enclose* in NARS[15,16] and APL2 as *box* in SAX[17] and J, and discussed in the section on atomic representations) makes it possible to box arguments of any rank and assemble them into a single argument for any function. In particular, it makes possible the use of such a boxed array as the argument to an indexing function, adopted in SAX and J and called *from*.

As may be seen,[18] the function *rotate* was initially defined so that the right argument specified the amount of rotation. The roles of the arguments were later reversed to accord with a general mnemonic scheme in which a left argument `a` together with a dyadic function `f` (denoted in J by `a&f`) would produce a "meaningful" monadic function. Exceptions were, of course, made for established functions such as *divided by*. The scheme retains some mnemonic value, although the commute adverb (`~`) provided in J and in SAX makes either order convenient to use. For example, `5 %~ 3` would be read as *5 into 3*.

In APL\360 it was impossible to define a new function within a program. This was rectified in APLSV[19] by defining a *canonical representation of a* function (a matrix `M` whose first row was a header, and whose succeeding rows were the sentences of the definition); a fix function `⎕FX` such that `⎕FX M` yielded the name of the function as an explicit result, and established the function as a side effect; and an inverse function `⎕CR`, which when applied to the name of a function produced its canonical representation as an explicit result. The ability to define ambivalent functions was added in a University of Massachusetts system,[20] and was soon widely adopted.

The function `⎕FX` established a function only as a side effect, but the scheme has been adapted to J by providing a *conjunction* (`:`) such that `m : d` produces an unnamed function that may be applied directly, as in `x m : d y`, or may be assigned a name, as in `f=.m : d`. See the section on name assignment.

Following an idea that Larry Breed picked up at a lecture by the late Professor A. Perlis of Yale, we adopted a scheme of *dynamic localization* in which names localized in a function definition were known to further functions invoked within it.

This decision made it possible to pass any number of parameters to subordinate functions, and therefore circumvented the limitation of at most two explicit arguments, but it did lead to a sometimes confusing profusion of names localized at various levels. The introduction of atomic representation (box and enclose) has made it convenient to pass any number of parameters as explicit arguments; in J this has been exploited to allow a return to a simpler localization scheme in which any name is either strictly local or strictly global.

Arrays

Perhaps because of the influence of a course in tensor analysis taken as an undergraduate, I adopted the notion that every function argument is an array, and that arrays may be classified by their *ranks;* a scalar is rank 0, a vector rank 1, a matrix rank 2, and so on.

The application of arithmetic (or scalar) function such as + and × also followed tensor analysis; in particular the *scalar extension,* which allowed two arguments to differ in rank if one were a scalar. In defining other functions (such as reshape and rotate), we attempted to make the behavior on higher-rank arrays as systematic as possible, but failed to find a satisfying uniform scheme. Such a uniform scheme (based on the notion of cells) is defined in "A Dictionary of APL,"[6] and adopted in SAX and in J.

A *rank-k cell* of *A* is a subarray of *A* along *k* contiguous final axes. For example, if:

```
   A
abcd
efgh
ijkl

mnop
qrst
uvwx
```

then the list *abcd* is a 1-cell of *A*, the table from *m* to *x* is a 2-cell of *A*, the atom *g* is a 0-cell of *A*, and *A* itself is a 3-cell of *A*.

Each primitive function has *intrinsic* ranks, and applies to arrays as a collection of cells of the appropriate rank. For example, matrix inverse has rank 2, and applies to an array of shape `5 4 3` as a collection of five 4 by 3 matrices to produce a result of shape `5 3 4`, a collection of five 3 by 4 inverses of the 4 by 3 cells.

Moreover, the rank conjunction (denoted in J by `"`) produces a function of specified rank. For example, the intrinsic rank of ravel is unbounded and (using the shape `2 3 4` array *A* shown above):

```
   ,A
abcdefghijklmnopqrstuvwx

   , "2 A
abcdefghijkl
mnopqrstuvwx
```

Further discussion of cells and rank may be found in the section on tacit programming, and in Reference 21.

The central idea behind the use of cells and a rank operator was suggested to me at the 1982 APL conference in Heidelberg by Arthur Whitney. In particular, Arthur showed that a reduction along any particular axis (`+/[`*I*`]`*A*) could be neatly handled by a rank operator, as in `+/"I A`. By further adopting the idea that every primitive possessed intrinsic ranks (monadic, left, and right) I was able, in Reference 6, to greatly simplify the definition of primitives: each function need be defined only for cells having the intrinsic ranks, and the extension to higher-rank arguments is uniform for all functions.

Adverbs and conjunctions

Even after tasting the fruits of generalizing the ∑ notation of mathematics to the form `f/` that permitted the use of functions other than addition, it took some time before I recognized the advantages of a corresponding generalization of the *inner* or *matrix* product to allow the use of functions other than addition and multiplication. Moreover, I thought primarily of the derived functions provided by these generalizations, and neither examined the nature of the slash itself nor recognized that it behaved like a Heaviside operator.

However, when we came to linearize the notation in the implementation of APL\360, the linearization of the inner product (which had been written as one function on top of the other) forced the adoption of a symbol for the conjunction (as in `M+.×N`). This focused attention on the adverbs and conjunctions themselves, leading to a recognition of their role and to the adoption of the term *operators* to refer to them.

In reviewing the syntax of operators we were disturbed to realize that the slash used for reduction applied to the (function) argument to its *left,* and even considered the possibility of reversing the order to agree with the behavior of monadic functions. However, Adin Falkoff soon espoused the advantages of the established scheme, pointing out that the adoption of a "long left scope" for operators would allow the writing of phrases such as `+.×/` to denote the function "inner product reduction," which might be applied to a rank-3 array.

We also realized that the use of the slash to denote compression (as in `1 0 1 0 1/'abcde'` to yield `'ace'`) seemed to imply that the slash was ambiguous, sometimes denoting an operator, and sometimes a function. This view was adopted in NARS and in the precursor to APL2. Alternatively, adverbs and conjunctions could be assumed to apply to both nouns and verbs, giving different classes of derived verbs in the different cases. In this view, compression was not a dyadic function denoted by the slash, but was rather the derived function resulting from the application of the adverb / to a noun.

The application of adverbs and conjunctions to nouns was adopted in SHARP,[22] SHARP/HP, SAX, and J, but was resisted in other dialects, in spite of the fact that the phrase `⌽[3]` for applying reversal on axis 3 furnished an example of such usage in early APL, and in spite of the implied use of nouns in Heaviside's notation D^2`f` for the second derivative of `f`.

In calculus, the expression `f+g` is used to denote the sum of functions `f` and `g`, that is, `(f+g) x` is defined as `(f x) + (g x)`. The utility of such constructs as `f+g` and `f×g` was clear, and I realized that they could be handled by operators corresponding to the functions + and × . What appeared to be needed was an adverb that would apply to a function to produce a *conjunction*. However, I was reluctant to complicate the grammar by introducing results other than functions from adverbs, and I began by suggesting, in Reference 23, a limited solution using composite symbols such as + overstruck by an overbar.

Somewhat later I discussed this matter with Arthur Whitney, and he quickly suggested an operator that we modified slightly and presented as the *til* operator in Reference 24, using the definition `x (f til g) y` is `(g y) f x`. The fork discussed in the section on grammar and order of execution now provides a more convenient solution, using expressions such as `f+g` and `f×g`.

In mathematics, the notions of inner product and outer product are used in rather limited areas. In APL systems, operators provide generalizations of them that not only broaden their uses, but make them more readily comprehensible to non-mathematicians. Much the same is true of "duals" in mathematics, but because the generalization of APL is not so widely known or used, it merits some attention here.

It is useful to view almost any task as performed in three phases: preparation, the main task, and undoing the preparation. In programming terms this would appear as `inversep main p argument`. In other words, the main function is performed *under* the preparation `p`.

In J the *under* conjunction is denoted by `&.` and is defined as follows:

```
   m&.p y is inversep m p y
 x m&.p y is inversep (p x) m (p y)
```

For example, since `^.` denotes the natural logarithm in J, the expression `a +&.^. b` yields the product of `a` and `b`. The *under* conjunction is commonly used with the function *open* (whose inverse is *box*) discussed in the section on atomic representations.

Name assignment

In mathematics, the symbol = is used to denote both a relation and the copula in name assignment (as in "let x=3"). In APL, the arrow was first used for the copula in Reference 18, and has been used in all dialects until the adoption of `=.` and `=:` in J.[21]

The use of the copula was initially restricted to nouns, and names were assigned to user-defined functions by a different mechanism in which the name of the function was incorporated in the representation to which the function ⎕*FX* was applied, as discussed in the previous section on functions. The use of the copula for this purpose was proposed in Reference 23, implemented in SHARP/HP, and later adopted in Dialog[25] and in J. These implementations provided for adverbs and conjunctions in the same manner. However, this use of the copula has not been adopted in other implementations, perhaps because the representations used for functions make its adoption difficult.

Indirect assignment was first proposed in Reference 26, and is implemented in J and defined in Reference 21. Two copulas are used in J, one for local assignment (`=.`), and one for global (`=:`) assignment.

Grammar and order of execution

Grammatical rules determine the order of execution of a sentence, that is, the order in which the phrases are interpreted. In Reference 3, the use of parentheses was adopted as in mathematics, together with the rule (Reference 3, page 8) that "The need for parentheses will be reduced by assuming that compound statements are, except for intervening parentheses, executed from right to left."

In particular, this rule implies that there is no hierarchy among functions (such as the rules in mathematics that power is executed before multiplication before addition). Long familiarity with this hierarchy occasioned a few lapses in my book,[3] but the new rule was strictly adopted in the APL\360 implementation. APL\360 also *introduced* a hierarchy, giving operators precedence over functions.

The result was a simple grammar, complicated only by the bracket-semicolon notation used for indexing. This was later complicated by the adoption, in most systems, of the *statement separator* (denoted by a diamond). The utility of the statement separator was later vitiated in some systems (including SHARP, SAX, and J) by the adoption of dyadic functions *lev* and *dex*, which yielded their left and right arguments, respectively.

The grammatical rules left certain phrases (such as a sequence of nouns) invalid. In NARS and in APL2 meanings were assigned to a sequence of nouns: if `a` and `b` are the nouns "hold" and "on," then the phrase `a b` yields the two-element list of enclosed vectors. The adoption of such "strands" led to a modification of the grammatical rules based upon left and right "binding strengths" assigned to various parts of speech, as discussed in References 27 and 28. In particular these rules required that the phrase `2 3 5[1]` be replaced by `(2 3 5)[1]`.

Other changes in grammar were adopted in J: the bracket-semicolon indexing was replaced by a normal dyadic verb *from*; and any isolated sequence of verbs was assigned a meaning based upon the *hook* and *fork*, first proposed in Reference 29 and briefly explained next. The result is a strict grammar in which each phrase for execution is chosen from the first four elements of the execution stack, and eligibility for execution is determined by comparison with a 14 by 4 parsing table as shown in Reference 21.

Because the hook and fork (as well as several other previously invalid phrases) play a significant role in the tacit programming discussed in a later section, they are further elaborated here. Briefly, if

```
   mean=.+/%#
```

then

```
   mean x
```

is equivalent to

```
   (+/x)%(#x)
```

The dyadic case is defined analogously. If

```
   diffsq=. +*-
```

then

```
   a diffsq b
```

is

```
   (a+b)*(a-b)
```

The hook and the fork may be expressed graphically as follows:

```
        FORK              HOOK
     g        g        g         g
    / \      / \      / \       / \
   f   h   f    h   y   h     x   h
   |   |  / \  / \      |         |
   y   y x  y x  y      y         y
```

Two further points should be noted:

1. A longer train of verbs will resolve into a sequence of forks and hooks. For example, `taut=. <:=<+.=` is equivalent to two forks, as in `taut=. <:=(<+.=)`, and expresses the tautology that *less than or equal* (`<:`) *equals* (`=`) *less than* (`<`) *or* (`+.`) *equal* (`=`).
2. In the expression `(+/ % #) 2 3 4 5` to produce the mean of the list `2 3 4 5`, the parentheses are clearly essential, since `+/ % # 2 3 4 5` would yield `0.25`, the sum of the reciprocal of the number of items. However, it must be emphasized that the parentheses perform their normal function of grouping, and are not needed to explicitly produce forks, as may be seen from the earlier examples.

Atomic representations

It is commonplace that complex constructs may be conveniently represented by arrays of simpler constructs: a word by a list of letters, a sentence by a list of words, a complex number by a list of two real numbers, and the parameter of a rotation function by a table of numbers, and so on.

However, it is much more convenient to use atomic representations, which have rank 0 and are therefore convenient to combine into, and select from, arrays. For example, the representation `3j4` used for a complex number in APL systems is an atom or scalar.

In Reference 30, Trenchard More proposed a representation scheme in which an *enclose* function applied to an array produced a scalar representation of the argument. This notion was adopted or adapted in a number of APL systems, beginning with NARS, and soon followed by APL2.

A somewhat simpler scheme was adopted in SHARP in 1982, was presented in "A Dictionary of APL"[6] in 1987, and later adopted in SAX and J: a function called *box* (and denoted by <) applied to any noun produces an atomic representation of the noun that can be "decoded" by the inverse function *open* (denoted by >) to yield the original argument.

A desire for similar convenience in handling collections of functions led Bernecky and others to propose (in References 31 and 32) the notion of *function arrays*. These have been implemented as *gerunds* in J by adopting atomic representations for functions.

Implementations

Because of a healthy emphasis on standardization, many distinct implementations differed slightly, if at all, in the language features implemented. For example, the IBM publication APLSV User's Manual[19] written originally for APLSV applied equally to VS APL and the IBM 5100 computer.

Despite the present emphasis on the evolution of the language itself, certain implementations merit mention:

1. The IBM 5100 mentioned above is noteworthy as one of the early desktop computers, and as an implementation based on an emulator of the IBM System/360* and a read-only memory copy of APLSV.
2. The I-APL implementation provided the first shareware version of APL, aimed at making APL widely available in schools.

Implementations representing the two main lines of development mentioned in the introduction are now discussed briefly. The first is the *nested array* system NARS conceived and implemented by Bob Smith of STSC and incorporating ideas due to Trenchard More[30] and J. A. Brown (Doctoral thesis, University of Syracuse). In addition to the *enclose* and related facilities that provide the nested arrays themselves, this implementation greatly expanded the applicability of operators. In the APL2 implementation, Brown has followed this same line of development of nested arrays.

Somewhat after the advent of NARS, the SHARP APL system was extended to provide *boxed* elements in arrays, as reported in Reference 22. New operators (such as the *rank)* were also added, but their utility was severely limited by the fact that operators were not (as in NARS) extended to apply to user-defined functions and derived functions. In the succeeding SAX and J implementations such constraints have been removed.

Tacit programming

A tacit definition is one in which no explicit mention is made of the arguments of the function being defined. For example:

```
      sum=. +/
      mean=. sum % #
      listmean=. mean"1
```

```
   [a=. i. 5
0 1 2 3 4

   sum a
10

   mean a
2

   [table=. i. 3 5
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14

   mean table
5 6 7 8 9

   listmean table
2 7 12
```

By contrast, definition in most APL dialects makes explicit mention of the argument(s):

```
      ⎕FX 2 7⍴'Z←SUM X Z←+/X'
SUM
```

Tacit programming offers several advantages, including the following:

1. It is concise.
2. It allows significant formal manipulation of definitions.
3. It greatly simplifies the introduction of programming into any topic.

Since the phrase `+/` produces a function, the potential for tacit programming existed in the earliest APL; but the restrictions on the copula prevented assignment of a name to the definition, and therefore prohibited tacit programming.

In any case, the paucity of operators and the restrictions that permitted their application to (a subclass of) primitive functions only, made serious use of tacit programming impossible. In later dialects these restrictions have been removed, and the number of operators has been increased.

I now provide a few examples of tacit programming in J, first listing the main facilities to be exploited. The reader may wish to compare such facilities in J with similar facilities defined by Backus[13] and by Curry.[33] For example, Curry's combinators W (elementary duplicator) and C (commutator) are both represented by the adverb ~ in J, according to the following examples:

`/:~b`	is	`b/:b`	(that is, a sort of `b`)
`a %~b`	is	`b%a`	(that is, `a` into `b`)

The facilities to be used in the examples include the *hook*, *fork*, and ~ already defined, as well as the following which, although defined in terms of specific verbs, apply generally. It may be necessary to consult Reference 21 for the meanings of certain verbs, such as `*:` (square), `%:` (square root), and `^.` (log). Five examples follow.

1.	`2 &^ y`	is	`2 ^ y`	(Called *currying*)
2.	`^ & 2 y`	is	`y^2`	(Called *currying*)
3.	`-&^. y`	is	`-^. y`	Composition
4.	`x -& ^. y`	is	`(^.x)-(^.y)`	Composition
5.	`x -@ ^ y`	is	`- x ^ y`	Atop

Some examples from statistics are shown next.

```
sum=. +/
mean=. sum % #
norm=. - mean
std=.%: & sum & *: & norm
```

Entry of a function alone causes a display of its definition, a display that can be captured and manipulated as a straightforward boxed array. Thus:

```
 std
┌─────────────────┬─┬────┐
│┌──────────┬─┬──┐│&│norm│
││┌──┬─┬───┐│&│*:││ │    │
│││%:│&│sum││ │  ││ │    │
││└──┴─┴───┘│ │  ││ │    │
│└──────────┴─┴──┘│ │    │
└─────────────────┴─┴────┘
```

In function tables, the `f` outer product of APL is in J the dyadic case of `f/`. For example:

```
   [a=. b=. i. 5
0 1 2 3 4

   a +/ b
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

   a*/b
0 0 0  0  0
0 1 2  3  4
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16
```

```
    a!/b
 1 1 1 1 1
 0 1 2 3 4
 0 0 1 3 6
 0 0 0 1 4
 0 0 0 0 1
```

Such a table can be made easier to interpret by displaying it with appended arguments, using the following tacit definitions:

```
    over=.({.,.@;}.)&":@.
    by=. (,~"_1 ' '&;&,.)~
    a by b over a !/ b
┌─┬─────────┐
│ │0 1 2 3 4│
├─┼─────────┤
│0│1 1 1 1 1│
│1│0 1 2 3 4│
│2│0 0 1 3 6│
│3│0 0 0 1 4│
│4│0 0 0 0 1│
└─┴─────────┘
```

Adverbs may be defined tacitly in a number of ways, as follows:

```
    sum \ a
 0 1 3 6 10

    scan=. /\
    + scan a
 0 1 3 6 10

    - scan a
 0 _1 1 _2 2

    table=. /(['by']'over')\
    2 3 5 *table 1 2 3 4 5
┌─┬──────────────┐
│ │1  2  3  4  5 │
├─┼──────────────┤
│2│2  4  6  8 10 │
│3│3  6  9 12 15 │
│5│5 10 15 20 25 │
└─┴──────────────┘

    a <table b
┌─┬─────────┐
│ │0 1 2 3 4│
├─┼─────────┤
│0│0 1 1 1 1│
│1│0 0 1 1 1│
│2│0 0 0 1 1│
│3│0 0 0 0 1│
│4│0 0 0 0 0│
└─┴─────────┘
```

Cited references and note

1. A. D. Falkoff and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development* 17, No. 4, 324–334 (1973)

2. A. D. Falkoff and K. E. Iverson, "The Evolution of APL," *ACM SIGPLAN Notices* 13, No. 8, 47 57 (1978)

3. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962), p. 16

4. See the 1971 Chelsea edition of Heaviside's *Electromagnetic Theory* and the article by P. Nahin in the June 1990 issue of *Scientific American*

5. R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney, "APL/?," *APL90 Conference Proceedings*, *APL Quote Quad* 20, No. 4, ACM, New York (1990)

6. K. E. Iverson, "A Dictionary of APL," *APL87 Conference Proceedings*, *APL Quote Quad* 18, No. 1, 202–211, ACM, New York (1987)

7. R. Hodgkinson, "APL Procedures," *APL86 Conference Proceedings*, *APL Quote Quad* 16, No. 4, ACM, New York (1986)

8. R. K. W. Hui, K. E. Iverson, and E. E. McDonnell, "Tacit Programming," *APL91 Conference Proceedings*, *APL Quote Quad* 21, No. 4, ACM, New York (1991)

9. P. C. Berry, *Sharp APL Reference Manual*, I. P. Sharp Associates, Toronto, Canada (1979)

10. K. E. Iverson, *Tangible Math*, Iverson Software Inc., Toronto, Canada (1990)

11. A. Camacho, "I-APL Status Report," *Vector: The Journal of the British APL Association* 4, No. 3, 8–9 (1988)

12. *APL2 Programming: System Services Reference*, SH20-9218, IBM Corporation (1988); available through IBM branch offices

13. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* 21, No. 8, 613–641, (1978)

14. A. D. Falkoff and K. E. Iverson, *APL\360 User's Manual*, IBM Corporation (1966)

15. R. Smith, "Nested Arrays, Operators, and Functions," *APL81 Conference Proceedings*, *APL Quote Quad* 12, No. 1, ACM, New York (1981)

16. C. M. Cheney, *Nested Arrays Reference Manual*, STSC Inc., Rockville, MD (1981)

17. *SAX Reference*, 0982 8809 El, I. P. Sharp Associates, Toronto, Canada (1986)

18. K. E. Iverson, "The Description of Finite Sequential Processes," *Proceedings of a Conference on Information Theory*, C. Cherry and W. Jackson, Editors, Imperial College, London (August 1960)

19. *APLSV User's Manual*, GC26-3847-3, IBM Corporation (1973)

20. C. Weidmann, *APLUM Reference Manual*, University of Massachusetts (1975)

21. K. E. Iverson, *The ISI Dictionary of J*, Iverson Software Inc., Toronto, Canada (1991)

22. R. Bernecky and K. E. Iverson, "Operators and Enclosed Arrays," *APL User's Meeting*, I. P. Sharp Associates, Toronto, Canada (1980)

23. K. E. Iverson, *Operators and Functions*, Research Report 7091, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1978)

24. A. T. Whitney and K. E. Iverson, "Practical Uses of a Model of APL," *APL82 Conference Proceedings*, *APL Quote Quad* 13, No. 1, ACM, New York (1982)

25. *Dyalog APL Reference Manual*, Dyadic Systems Ltd., Alton, Hants, England (1982)

26. K. E. Iverson, "APL Syntax and Semantics," *APL83 Conference Proceedings*, *APL Quote Quad* 13, No. 3, 223–231, ACM, New York (1983)

27. J. P. Benkard, "Valence and Precedence in APL Extensions," in *APL83 Conference Proceedings*, *APL Quote Quad* 13, No. 3, ACM, New York (1983)

28. J. D. Bunch and J. A. Gerth, "APL Two by Two-Syntax Analysis by Pairwise Reduction," *APL84 Conference Proceedings*, *APL Quote Quad* 14, No. 4, ACM, New York (1984)

29. K. E. Iverson and E. E. McDonnell, "Phrasal Forms," *APL89 Conference Proceedings*, *APL Quote Quad* 19, No. 4, ACM, New York (1989)

30. T. More, Jr., "Axioms and Theorems for a Theory of Arrays," *IBM Journal of Research and Development* 17, No. 2, 135–157 (1973)

31. R. Bernecky, "Function Arrays," *APL84 Conference Proceedings*, *APL Quote Quad* 14, No. 4, ACM, New York (1984)

32. J. A. Brown, "Function Assignment and Arrays of Functions," *APL84 Conference Proceedings*, *APL Quote Quad* 14, No. 4, ACM, New York (1984)

33. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. 1, North Holland Publishers, Amsterdam, Netherlands (1968)

Kenneth E. Iverson, *44 Charles St. West, No. 4709, Toronto, Ontario M4Y 1R8, Canada.* Dr. Iverson received a B.A. in mathematics and physics from Queen's University, Kingston, Canada in 1950, an M.A. in mathematics in 1951, and a Ph.D. in applied mathematics from Harvard University. He was an assistant professor at Harvard from 1955 to 1960. From 1960 to 1980 he was employed by IBM Corporation's Research Division where he became an IBM Fellow in 1970. After leaving IBM in 1980, Dr. Iverson was employed by I. P. Sharp Associates until 1987. He has received many honors, in addition to becoming an IBM Fellow, including the AFIPS Harry Goode Award in 1975, the ACM Turing Award in 1979, and the IEEE Computer Pioneer Award in 1982. He is a member of the National Academy of Engineering in the United States. Currently he is working on J and the use of J in teaching.

•