As this is my third attempt to explain what I have done with my professional life, the following quotation, which I have used before, from one of my favourite fictional characters, takes on an added significance:

> *"So there it is", said Pooh, when he had sung this to himself three times. "It's come different from what I thought it would, but it's come. Now I must go and sing it to Piglet."* ■

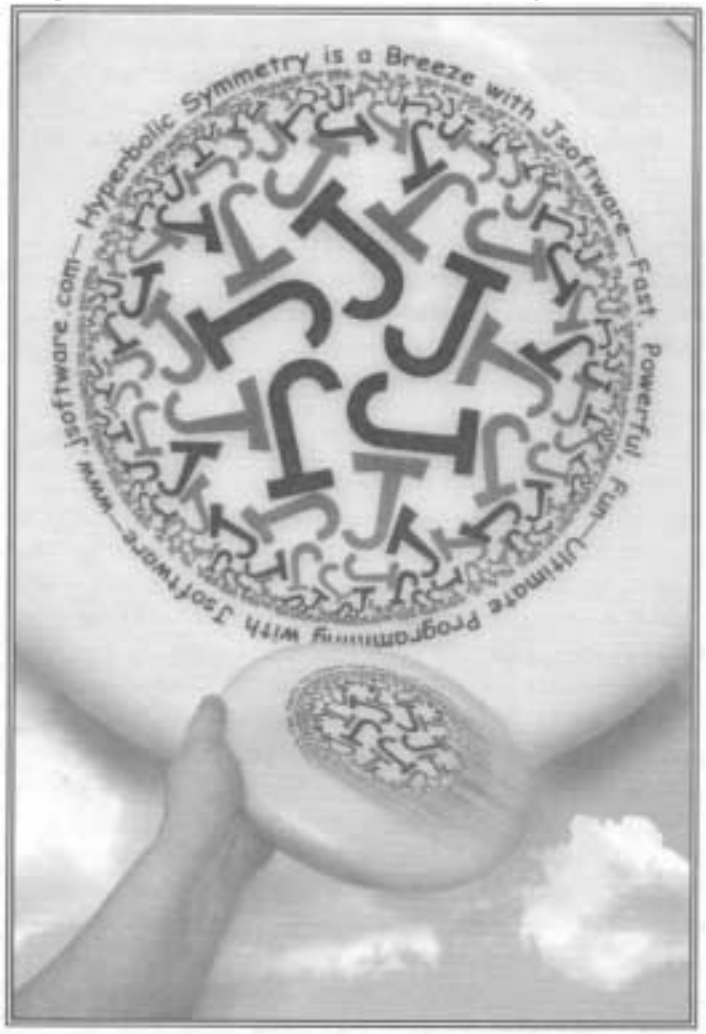
Acknowledgements

I wish to thank the following persons for their very helpful comments on earlier versions of this paper: Kenneth Iverson, Howard Peelle, Clifford Reiter, and Alison Smillie.

***Keith Smillie** is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2H1. His e-mail address is "smillie@cs.ualberta.ca".*

Non-Sequitur

The J Frisbee —*Courtesy of* ***Cliff Reiter***



APL In the New Millennium

—by ***Kenneth E. Iverson***
Toronto, Ontario

SINCE IBM's APL\360 BECAME AVAILABLE IN 1966, many dialects have been developed, and competition has led to emphasis on their differences, an emphasis reflected in their distinctive names: APL\1130, APL\360, APLSV, APL2, SHARP APL, Nial, Dyalog APL, A, APL2000, J, K, and others.

Although natural to healthy competition, the emphasis on differences has discouraged the sharing of ideas, and still tends to blind programmers to the ease of moving between dialects, an ease not shared by programmers unschooled in the core ideas of APL.

As emphasized in [1], these core ideas were:

- The adoption from Tensor Analysis of a systematic treatment of arrays, in which every entity is an array, and different *ranks* lead to *scalars*, *vectors* (or *lists*), *matrices* (or *tables*), and *higher-dimensional arrays* (or *reports*).
- Operators (in the sense introduced by Heaviside [2]), which apply to functions to produce related functions.

In this paper I will review developments in the APL dialects, emphasizing similarities and the ways in which competing ideas have been, and could be, shared and adapted to competing systems. My hope is to encourage the relatively small APL family to mute their differences, and present a more united face to the programming world.

Alphabets

Although the particular alphabet, or even the font used, is a most superficial aspect of a language, it can make a dramatic assault on a beginning reader—as anyone who first met German in the Gothic font can testify. First encounters with the unfamiliar alphabet of the earliest APL has certainly discouraged many, in spite of its highly-mnemonic character.

At the time of its design there was no adopted standard, and it seemed reasonable to exploit the newly available IBM Selectric typewriter (with its easily-changed *typeball*) to design our own alphabet, and to use the backspace ability of the typewriter to produce composite (*overstruck*) characters.

The APL community was too small to influence the design of the now widely-used ASCII alphabet, and our use of special characters led to a series of unforeseen difficulties that have significantly inhibited the use of APL:

- When the "glass terminal" provided by the cathode ray tube supplanted the typewriter, it was incapable of backspacing to provide the composite characters of APL.
- APL characters were not provided by early printers, and there was a considerable delay before specialized alphabets could be downloaded to them.

- Such difficulties have led some dialects (such as Nial) to adopt ordinary names as *reserved words*, an approach that the special characters had allowed us to avoid.
- Use of the internet also poses problems, because APL characters are not generally handled by browsers.

J and K use only the ASCII alphabet, but yet avoid the use of reserved words. K uses mainly single-character non-alphabetics, and J supplements these by a scheme that uses a suffixed period or colon. For example, < denotes *less-than*, <. denotes *lesser-of* (minimum), and =: denotes assignment.

It would, of course, be impractical for any APL system to switch to a rival alphabet, and this discussion is meant only to suggest supplements based on rival ideas. For example:

1. Many ASCII symbols (such as @, &, ^, and %) go unused in most APL systems, and could be used in various ways:

   One or two might be used as suffixes, as in <@ or nub@ as names for primitives.

   The * adopted for power in APL\360 has since become universally recognized as the symbol for multiplication, and might better be replaced by the ^, as first proposed by De Morgan in mathematics, and since adopted in some non-APL languages.
2. The percent symbol (which signifies "divided by 100") has been adopted for division by some non-APL systems, and could be more widely adopted in APL. After all, the APL symbol for division is no longer familiar in math.

Number representation

APL\360 introduced the useful representation of a negative number distinct from the negation function (-) that might produce it. This has been continued in all APLs in forms that vary from the special overbar symbol used in APL\360. Constrained to ASCII symbols, J uses the underbar, and K uses juxtaposition: _3 for a negative number, as contrasted with - 3 for negation.

APL systems use the exponential notation adopted from Fortran, some using the uppercase E, and some the lowercase. The notion has been extended to other kinds of numbers, as in 2d45 for a complex number in polar representation (APL2); 3j4 for a complex number (SHARP APL); and 2r3 for a rational number (J).

Grammar

The grammar (parsing rules) of APL\360 were simple and relatively uniform, with no precedence among functions, but with operators given precedence over functions.

In particular, the relative positions of functions and arguments were fixed; for example, factorial n was written as !n, *not* n!. There are, however a few characteristics that merit comment.

Name assignment

A left arrow was used to assign names to arrays of numbers and characters, but a quite different mechanism was used for assigning names to functions, and there was no provision for defining operators.

In APL2, operator definition was introduced by extending the system for function definition to allow further parameters. Most systems have adopted this scheme, but J uses a single copula (the =: mentioned earlier) for all three cases, and Dyalog APL uses it for two.

Valence

APL\360 systematically adopted the double use of symbols from the scheme suggested by subtraction (3-2) and negation (-2) in mathematics, calling the two cases *dyadic* and *monadic*. For example, !n denotes the factorial, and m!n denotes the related binomial coefficients.

Most systems (with the exception of Nial) continued this *ambivalent* use of primitives, but did not extend it to the *derived* functions produced by operators. In J, *all* functions are ambivalent. For example, +/ y denotes *sum over* y, and x +/ y denotes the addition table; that is, the plus-outer-product denoted by x ∘.+ y in APL\360 and APL2.

Indexing

Because of the need for multiple index arguments for a multidimensional array, APL\360 adapted from Fortran the notation A[I;J;K], departing from the normal form for a dyadic function. In particular, it was not possible to assign a name to the complete index argument.

The introduction of *nested arrays* in APL2 made possible the treatment of a set of indices such as I;J;K as a single entity. However, many APL systems did not fully exploit this to rationalize indexing.

APL\360 introduced a further anomaly in indexing by providing either 1-origin or 0-origin indexing, set originally by a *system command* of the form)ORIGIN, and later by a *system variable*, ⎕IO.

This choice of index origin has been maintained in most systems, but J is restricted to 0-origin. Since indexing in J is a normal dyadic function, an operator can be used to modify it, as well as the related index generator analogous to the iota of APL\360.

The restriction to 0-origin has simplified the introduction of *negative indexing*, with indices for n items running from negative n to n-1.

The *indexed assignment* A[I;J]←M is a further convenient (though grammatically anomalous) scheme introduced in APL\360. It is maintained in most systems, although the three essential arguments can be handled by an *amend* operator, such as the } used in J in the form: M ind } A.

Terminology

Coming from a common background in math, we naturally adopted mathematical terminology in APL\360, in spite of the facts that:

- The term *operator* (used in the sense of Heaviside) conflicts with its common use in elementary mathematics as a synonym for function.
- The term *variable* used for a name assigned to a quantity suggests that its meaning might *vary* due to possible reassignment. But the same possibility exists for defined functions, and the terms *variable* and *function* do not adequately reflect the possible cases.

Terms from English grammar can provide the necessary distinctions, using the close analogy between *function* and *verb* as denoting *actions*, together with the *nouns* and (variable) *pronouns* on which they act.

Moreover, *adverbs* are analogous to *operators* (such as /) that modify a single verb, and *conjunctions* (such as the copulative conjunction *and* in the phrase "run and hide") are analogous to operators (such as the inner-product) that apply to two verbs.

Finally, the familiar English terms *list*, *table*, and *report* are more commonly understood (and are fully as accurate as) the terms *vector*, *matrix*, and *higher-dimensional array*.

Opportunities

Most APL systems share unexploited cases that may be introduced without conflict. We will illustrate these opportunities by three classes.

Complex arguments

Although complex numbers serve primarily in mathematical expressions, their two parts (real and imaginary) can be convenient in functions that require the specification of two independent parameters.

For example, a format function `F` might be defined so that `12j3 F x` formats `x` with a width of 12 spaces and with 3 digits following the decimal point. A list of such complex arguments could be used to specify each column of the format of a table `x`.

Vector arguments

In APL\360, the expression `⍳5` produced a list of five successive integers, but the domain of iota did not include a vector argument. A useful non-conflicting extension could be defined to apply to a list of n integers to produce an array of successive integers of rank n (as in J) or a nested array of indices of an array of the same rank (as in Dyalog APL).

Trains

In calculus, the expression `f+g` is sometimes used to define a function equivalent to the sum of the functions `f` and `g`, and `f*g` is used for their product. Moreover, such a train of three functions is treated as an error in most APL systems, and could therefore be introduced without conflict.

Any three functions may be used. For example `+/%#` (the sum divided by the number of elements) defines the *mean* function, and `f,g` defines a function that catenates results, as in `+/,*/`.

More generally, a train of any odd number of functions defines a function, the last three defining a function as stated above, and this function entering into a similar definition with the remaining train.

For example, the identity function followed by - (subtraction) followed by the preceding three-element train for the mean defines the function "center on the mean."

Extensions

Functions and operators new to any system can of course be adapted from other systems without conflict. In early systems the choice of symbols posed a problem, soon solved in a general way by the adoption of a class of "quad names"; alphabetic names preceded by the quad character. This solution appears to continue in systems that adhere to the special APL character set.

We will now discuss a few of the many functions and operators that are candidates for adoption.

GCD and LCM

The logical *or* and *and* could be construed as special cases of *maximum* and *minimum* (when restricted to the Boolean domain 0 and 1), or as special cases of *greatest common divisor* (GCD) and *least common multiple* (LCM).

E. E. McDonnell noted that only the latter functions possessed the same identity elements as *or* and *and*, and ensured that the logical functions were extended to GCD and LCM in the SHARP APL system. The same extension could be made without conflict in any APL system.

Repeatable random numbers

In debugging a program it is often useful to generate "random" arguments in a repeatable manner. This can be done by resetting the random seed. It is more convenient to provide a random number generator (denoted, perhaps, by `?.`) that resets the seed on each use.

Variants

APLSV used a *system variable* to specify the comparison tolerance to be used in relations such as < and =. Such control can be made more convenient by a variant operator, as in `= VAR 0` for exact comparison.

A more interesting opportunity for variants occurs in the case of the *rising* and *falling* factorial functions, defined by `*/x+s*⍳n`, for `s` assigned the values `1` and `_1`, respectively. Moreover, a zero value for `s` gives the function `x` to the power `n`, and these functions (including the useful case of non-integer values for `s`) can all be treated as variants of the power function.

Ties

The sum `a+b+c+d+e` can be written as the reduction `+/a,b,c,d,e`. Moreover, the continued fraction `a+b/c+d/e` might lend itself to a similar reduction, except that it requires the *two* functions of addition and division.

Such a pair of functions (or any number) can be provided by a `TIE` operator, to be used in the form `+TIE% / a,b,c,d,e`.

The result of a tie can be used in other ways, as with a *case* operator, in which `f TIE g TIE h CASE i` uses the index produced by the function `i` to select one of the functions for execution. In particular, the tie of two functions can be used to make a recursive definition.

Universal sorting

A strict ranking can be imposed on *all* arrays (including characters, real and complex numbers, and open and nested arrays of any rank) so that sorting can be applied to *anything*. An example of such ranking is provided in J, and could be adapted to any system.

The workspace

When first proposed by Adin Falkoff for APL\360, the (32K) workspace provided a convenient and efficient organization of memory. However, the fixed size, and other characteristics of the workspace, have come to inhibit the use of APL.

In particular, the workspace organization did not facilitate the exploitation of the memory management offered by later machines and operating systems. Arthur Whitney made the first step in employing these facilities in his A system, and used text (*script*) files rather than workspaces.

The advantages of text files are beginning to be recognized. In an item on page 55 of the April issue of *Vector* (Vol. 16, No. 4), Ansii Seppala is quoted as follows: "the more I can work with text files, the easier it is—I am no longer a fan of the APL workspace." ■

Acknowledgment

I am indebted to Chris Burke for many helpful comments, particularly for his suggestion to discuss the matter of the workspace versus script files.

References

[1] Iverson, K. E., A Personal View of APL, *IBM Systems Journal*, Vol 30, No. 4, 1991

[2] Heaviside, Oliver, See the 1971 Chelsea Edition of Heaviside's *Electromagnetic Theory* and the article by P. Nahin in the June 1990 issue of *Scientific American*

***Kenneth E. Iverson** can be reached via e-mail at "kei@interlog.com" or by phone at 1-416-924-0007.*

With J

"A Palette Editor, Dad's First GUI"

*—by **Cliff Reiter***
Department of Mathematics
Lafayette College

GRAPHICAL USER INTERFACES (GUIs) have become a standard for programming. As a mathematician, I am usually content with functional programming, but I have had a couple of near GUI experiences and admit that good GUI design can offer significant productivity gains and enhance creativity in some contexts. My first near GUI experience was to update the J based GUI developed by a coauthor [1] in a release of J before the form editor was available, but that doesn't really count. Closer still, a couple of years ago I collaborated with my two oldest sons on the paper "Word of Words from Iterated Function Systems" [4]. That work allows a user to enter a word into a form, see Figure 1, and the result is a fractal copy of the word made up of fractal copies of the word. Figure 2 shows the code used upon the word "GUI". The script is available at [6]. At that time, one of those sons, Zach, was in elementary school while my eldest, Ben, was in middle school. The collaboration was real. I wrote the paper and the core code that composed gerunds; I also described two letters, "D" and "E", in terms of linear and polar deformations. Ben and Zach completed descriptions of the rest of the alphabet (a daunting task) and implemented the GUI. They spent far longer than I did working on the project. I critiqued an early version of the GUI and reached for the mouse

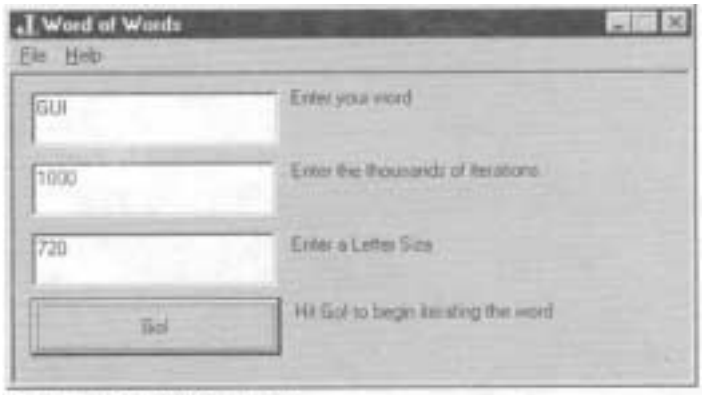


Figure 1: Word of Words Form

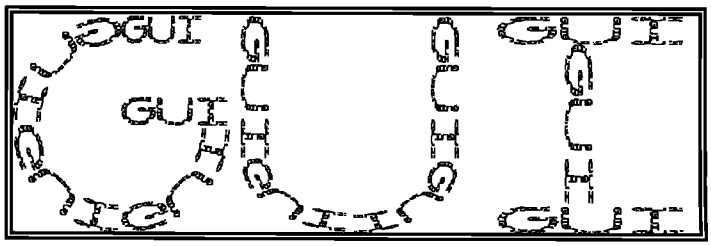


Figure 2: Fractal GUI