JUNK CONSIDERED HARMFUL

Open letter to the APL community --

A frequent criticism leveled against APL is that APL programs are esoteric and unreadable. For years many of us have been trying to convince people to the contrary through education, writing, and promotion. At the same time, we ourselves are perpetuating and emphasizing the belief in the hermeticism of APL with the programs published in APL Quote Quad. Quote Quad has printed some of the worst examples of APL programming I've ever seen. Their very appearance here implies to readers, both sympathetic and hostile, that these examples are good and reasonable and something to be proud of.

If the algorithms in Quote Quad are considered to be reasonable or even exemplary instances of APL programs, it is hardly surprising that some people refuse to take APL seriously. Furthermore, if Quote Quad publishes the best of what is available, I shudder to imagine what the rest looks like.

While these comments are critical, I do not intend them to be directed to any particular set of individuals. To the extent that a problem exists in APL style, the burden of guilt must be borne by all of us. In spreading the good word about APL, we have apparently neglected to emphasize the main point: programs must be readable by people. Unless we can impose some simple aspects of stylistic programming on the process of using APL, we might as well throw away our typespheres and go back to COBOL.

I would like to offer what I think are some commonsense suggestions to improve the readability of APL programs, particularly those destined for publication.

1. Use comments. The lamp symbol ⍝ was put into the language to allow program writers to include illuminating text. Such text can contain explanations of what the program does, restrictions on the arguments or environment, a record of changes, names of global objects, and so forth. Why is it that most functions in Quote Quad have no comments at all?

2. Structure. No process is so complex or monolithic that it can be expressed only by a function 188 lines long. Even 40 lines is often excessive, especially in an expository context such as a publication. It is easy to write subfunctions in APL. Long programs should be structured so that the reader can follow the flow. Where a longer function is required, sections of the program should be clearly delimited with comments explaining each part.

3. Flow of control. Control flow in APL is typically expressed by function calls or by branching. As mentioned above, better use should be made of functions. Where branches are necessary, all branch destinations should be labels. There is no excuse whatever for absolute branches to line numbers in APL programs. Conditional branch statements can be rendered more readable by use of simple functions such as IF. Similar functions can be used to express testing and reaction to exception or error conditions.

   One often-used "justification" for abstruse branching conventions is the "efficiency" of a particular APL implementation. If that is a problem in your installation, please don't burden the rest of us with it. If absolute branches or funny conditionals turn you on, write a function to eliminate labels and "compile" your IF's. But please don't subject readers of your source program to the agony and ambiguity of reading that kind of nonsense.

4. Naming. I know of no contemporary APL system that limits users to one-character identifiers. Nonetheless, most functions published in Quote Quad use nothing but. APL allows longer names. Restrictions to six-character identifiers disappeared with FORTRAN II. Especially in published functions, use of well-chosen names with mnemonic value enhances readability.

5. Implementation dependency. In production programs, it would be folly not to take advantage of the unique or proprietary features of your host APL system. In exposition of algorithms destined for information interchange or education, such use is self-defeating. The absence of language standards does little to simplify the task of preparing a program for publication. Even so, published programs should be "sanitized" to the extent that they are readable by APL'ers using a system other than your own. If it is necessary to use the rare undocumented private feature, it is common courtesy to include an explanation of its use and effect as comments in the program.

6. Gluing. Gluing is the practice of putting multiple, generally related, statements on the same line using artificial means. Though the stylistic motivation to group related short expressions is admirable, the result is an unreadable function. What's worse is that gluing is often dependent on the order of execution, rendering a statement not only unreadable but thoroughly ambiguous. In the absence of an accepted statement separator in APL, use of functions such as those below will yield the desired effect and make the programs readable.

```
    ∇ DO STMTS;FUNC
[1]  ⍝ STMTS IS A CHARACTER VECTOR
[2]  ⍝  OR MATRIX OF STATEMENTS.
[3]  ⍝ DO  TRANSFORMS STMTS INTO A
[4]  ⍝  FUNCTION 'FUNC' AND EXECUTES.
[5]  ⍝ ORIGIN INDEPENDENT
[6]  ⍝ GLOBALS: THEN (FN)
[7]   ⍎⍕FX 'FUNC' THEN STMTS
    ∇
```

```
    ∇ R←S1 THEN S2;DIM
[1]  ⍝ S1 IS A CHARACTER VECTOR.
[2]  ⍝ S2 IS A CHARACTER VECTOR
[3]  ⍝  OR MATRIX.
[4]  ⍝ BOTH REPRESENT STATEMENTS.
[5]  ⍝ THEN  RETURNS A CHARACTER
[6]  ⍝  MATRIX WITH S1 PRECEDING S2.
[7]  ⍝ ORIGIN INDEPENDENT
[8]  ⍝ GLOBALS: NONE
[9]   DIM←(⍴S1)⌈¯1↑⍴S2
[10]  →L IF 2=⍴⍴S2
[11]  S2←(1,⍴S2)⍴S2
[12] L:R←(DIM↑S1),[⎕IO]((1↑⍴S2),DIM)↑S2
    ∇
```

Then instead of such glued statements as

```
⎕PW←120×⎕IO←Z←1,0⍴R←⍳10
```

we can write a readable version:

```
DO '⎕PW←120' THEN '⎕IO←Z←1' THEN 'R←⍳10'
```

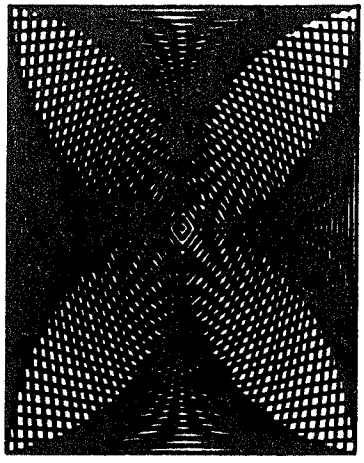
7. Side effects. One of the most insidious sources of programming errors as well as unreadability is the dependence on side effects. There are two kinds of side effects: those that are linguistically correct and those that derive from peculiarities of an implementation. The first class includes modification of non-local names through assignment or functions such as ⎕FX or ⎕EX. The second class is exemplified by what I call "pornography" -- dependence on the order of execution.

Astonishingly, it is still a matter of debate whether statements in APL, as a formal language, should have a defined order of execution, and if so, what it is. As a practical matter, however, there are variances in execution order among the several dozen currently used APL processors. Hence, as with the other points discussed in this letter, published algorithms should not include statements or expressions for which the meaning depends on the order of execution. In general, this includes any expression in which a variable is assigned and is also used outside the scope of the assignment subexpression as, for instance, Q×Q←R+2. Such expressions should be decomposed on several lines, to indicate ordering dependencies explicitly. Alternately an artifice such as the *DO* and *THEN* functions above should be used.

The suggestions made here are hardly an exhaustive survey of how to write readable programs. They simply point out some of what I consider to be the most blatant stylistic blunders that have found their way into the APL literature. While I propose these ideas primarily as a way to enhance the quality of published algorithms, they are also applicable in real programming.

If we wish to win the respect of professionals outside the APL community and maintain our own self-respect, we must clean up our own publications.

Philip S. Abrams
Vice President - Development
Scientific Time Sharing Corporation
7316 Wisconsin Ave.
Bethesda, MD 20014



CBS WAS READY AND WAITING...

BUT DELEGATE PROFILES UNNEEDED

NEW YORK - Had last week's Democratic National Convention been a contest like the Republicans' is going to be, CBS-TV could have projected the voting behavior of every delegate with a computer data base it compiled weeks before the convention even started.

The CBS Election and Survey Unit interviewed every convention delegate and stored those profiles in a data bank accessible via portable terminals near the convention floor, according to Warren Mitofsky, director of the unit.

CBS has been using computers for years to help bring viewers more detailed coverage of national conventions, but this is the first time its computer also has been programmed to count roll call votes, he said.

For this function, CBS' IBM 370/155 recorded and tabulated votes and then output the results to a minicomputer-driven character generator which projected the numbers right on viewers' TV screens.

To build the data base, CBS interviewers contacted every convention delegate and asked each a series of about 40 questions. News analysts wanted to know delegates' demographic characteristics --race, sex and nationality, for example--as well as their positions on the issues, Mitofsky said.

Since last week's chief unknown concerned the selection of a vice-presidential candidate, "we also wanted to know which vice-presidential contenders the delegates supported and which candidates they could not support," he said.