STYLE AND LITERACY IN APL

Michael Berry
Analogic Corporation
1 Audubon Rd., Wakefield, Massachusetts 01880, USA

Roland Pesch
I. P. Sharp Associates
220 California Ave., Palo Alto, California 94306, USA

...then the strut changed to the restless walk of a caged madman, then he whirled, and to a clash of cymbals in the orchestra and a cry of terror (perhaps faked) in the gallery, Mascodagama turned over in the air and stood on his head.
--Nabokov, Ada

"See me jump," said Dick.
"Oh, my! This is fun.
Come and jump.
Come and do what I do."
--Gray et al., The New Fun with Dick and Jane

There is a persistent belief in the APL community, reflecting perhaps some of the prejudice against APL outside that community, that "good style" in APL involves writing very short statements, using as few primitives as possible in each. It is easy enough to find an example in a discussion of APL in a general computing magazine:

> Mathematicians and engineers love APL for its conciseness and power, but there's quite a price to pay: APL programs are almost unreadable. It's very easy to write a single-line program that would take an average APL programmer a good fifteen minutes to figure out....Typically, good APL programmers write one line of comment for every line of code and try to keep their program lines short. [1]

Such opinions don't spring forth full-grown from the forehead of Zeus; they have their origins in the APL community (where else would an author, who does have his objective facts about APL straight, go for information about APL?).

For example, a recent issue of *APL Quote Quad* carried a "Style Guide" for functions submitted to the journal, including this: "VERTICAL: Write each function vertically with several short lines, rather than horizontally with long lines" [2]. Another example: an otherwise admirable document on APL programming standards, circulated in the San Francisco area, contains an appendix that introduces two functions by saying "The following are two functions which do the same thing. The first is a one liner which is pornographic. The second uses the same code but broken down into more readable pieces." [3] We will examine those two functions later in this paper.

First, let's explore this belief. Short function lines seem to be a crucial point. That must mean, for example, that it should be considered better to write

```
S←+/⍵
C←⍴⍵
Z←S÷C
```

than

```
Z←(+/⍵)÷⍴⍵
```

to calculate an average.

The justification is that the short statements are easier to read. This claim has gone unexamined for far too long. We will consider it from two perspectives:

In what way *is* a multiplicity of short statements easier to read than a single, longer one?

For *whom* is this style in general easier to read?

The first question is no doubt easy to answer. After all, if there's only one primitive (and assignment) in an expression, there can't be too much doubt about what that expression does, right? In `C←⍴⍵` above, for instance, `C` is clearly the number of elements in `⍵`. Whereas in a longer, more complicated expression, we might be daunted by the large number of funny squiggles, and never notice the `⍴⍵` buried in there somewhere.

But does this really get us much farther in finding out what's going on? We're calculating an average, after all-- there are two other statements involved. Let's see: the first statement, `S←+/⍵`, is just as easy: we're adding up the elements in `⍵`. Oh, and giving them the name `S`. Hope that's not too complicated, two things at once; pity we couldn't break it up further. Only one to go. Well, `Z←S÷C` is quite easy. Nothing to it either. We're dividing `S` by `C`, of course. But, wait-- what was it we were doing? Let's see, `S` was a sum, and `C` was,

hold on a minute while we check, a count, that's right. So we're dividing the *sum* of things in ⍵ by the *number* of things in ⍵-- of course, this must be an average. There, we're done. Easy, wasn't it?

Go ahead, laugh. We know, you don't have trouble keeping three statements straight in your head. But when you're done laughing, think about the general claim in the light of this example. While breaking up a line into short segments does indeed produce a program whose lines are easier to read than the original, that isn't the appropriate unit of comparison: the *lines* in the fragmented version do nothing of interest. Comparing the two programs is more interesting-- and here, the shoe is on the other foot. The fragmentation of thought, and the introduction of extra names into the calculation, makes it *harder* to keep track of what's going on, because to determine the meaning of the final result you must be aware of definitions that exist only in the immediate context, whereas in the brief and obvious (+/⍵)÷⍴⍵ everything is out where you can see it, and no reference is necessary to other parts of the expression. Consider again the example paraphrased into English:

Instructions for computing an average (Version 1)
Add up a bunch of numbers and divide by how many you had.

Instructions for computing an average (Version 2)
Add up a bunch of numbers, and call the result "Ess".
Count the numbers you added up, and call the result "Cee".
Divide Ess by Cee.

But why stop here? English is an even richer language than APL; it can probably provide even, ah, clearer versions of the instructions:

Instructions for computing an average (Version 3)
1. Some instructions follow.
2. The instructions are about to begin.
3. You have a list of numbers.
4. Reserve a spot called "Ess" to put sums in.
5. Put a zero in Ess.
6. Is there at least one number in the list?
7. If not, go to step 12.
8. Name the first number in the list "Ex".
9. Add Ex and Ess, and call the result "Ess" now.
10. Remove the first number from the list.
11. Go back to step 6.
12. . . .

We think we'll spare you (not to mention ourselves) the rest. Actually we took some liberties there; that could be a lot clearer yet.

So. Of versions 1, 2, and 3 of the *Instructions*, which do you think needs comments most? Maybe there are people who disagree, but our own feeling is that we'd definitely need to accompany 3 with some descriptive text (suppose we hadn't said what these instructions were meant to achieve?), probably some such text would also be handy for 2, and 1 needs it not at all. We assume none of our readers would actually prefer version 3, so let's leave it aside for now. But anyone who likes his APL expressions sold short (would you believe that was a typo and we meant "told"? No? Oh well) presumably feels utterances such as version 2 of the *Instructions* above are somehow more natural for human beings. We can only recommend experiment. We'd suggest the following one: next time you want a raise, be clear. Don't tell your boss something confusing like "I want another thirty thousand dollars". Say, "Call my salary 'Ess'. Call thirty thousand dollars 'Tee'. Add Ess and Tee, and call the result 'Ess'. Thanks!" Who knows, it may work better that way. But we doubt it would be because of greater clarity...

The intermediate names are only part of the story, though. Version 3 of the *Instructions* was not meant just to amuse us; it illustrates how easy it becomes to lose track of what's going on when a process is broken down into ridiculously tiny steps.

It also illustrates a more dangerous aspect of the "short and vertical" style of APL programming: carried just a little farther, it becomes scalar thinking-- conditioning a writer of APL to use this style risks encouraging inappropriate, inefficient use of the language as a tool.

The following function comes from an application in a real business environment:

```
    ∇ SINGLEPREMS
[1]   RPT←56 8⍴0
[2]   CHARGES←'C',0⍴MATACE←98,0⍴LOAD←0,
    0⍴MODE←1,0⍴MATAMOUNT←1000,0⍴LUMPSUM←0,
    0⍴RATING←1,0⍴NEWCORR←'Y',0⍴OPT←'B',
    0⍴CVINT←0.11
[3]   FACE←50,0⍴STATUS←'N',0⍴SEX←'M'
[4] L0:AGE←20
[5] L1:PREMAGE←AGE+1
[6] L4:RPT[(AGE-19);(+/FACE≥0 99)+
    (4×SEX='F')+2×(STATUS='S')]←GPCALBA
[7]   →((AGE←AGE+1)≤75)/L1
[8]   →((FACE←FACE+50)≤100)/L0
[9]   →(STATUS='S')/L2
[10]  STATUS←'S'
[11]  FACE←50
[12]  →L0
[13]L2:→(SEX='F')/L3
[14]  FACE←50,0⍴STATUS←'N',0⍴SEX←'F'
[15]  →L0
[16]L3:
      ∇
```

What's wrong with this function? Why, in one sense, nothing at all: it ran, it gave the proper answers, its author was happy with it for a long time. In another sense-- well, would you like to maintain it? Or would you rather deal with something like the following:

```
    ∇ RPT←SINGLEPREMS CVINT;AGE;MATACE
[1]   RPT←0 8⍴''
[2]   MATACE←98 ⍝ GLOBAL USED BY GPCALBA
[3]   AGE←20
[4] L1: RPT←RPT,[2]AGE GPCALBA CVINT
[5]   →(75≥AGE←AGE+1)/L1
    ∇
```

There are, of course, a number of differences between the two, and some of them depend upon examination of the subfunction *GPCALBA* (not shown here). One thing leaps immediately to one's notice:

while there *were* a couple of fairly long lines in the first version, the second was certainly not produced by breaking them up! The two functions *are* equivalent; it took careful reading of *GPCALBA*, a fairly involved function, to notice that there was no need at all for the nested loops and associated extra parameters in *SINGLEPREMS*-- all the calculations were (or could be made) parallel for all cases, and all cases were always covered. Had the consultant who did the work not been literate in APL, he could scarcely have made this simplification. Perhaps more important-- the original form arose because *scalar thinking is pervasive*. People's approach to problems is conditioned by their habits. The first version of *SINGLEPREMS* was written by an author not fluent in APL, who found it more congenial to address problems bit by bit. It reflects scalar thinking in its style: despite having middling long lines 2 and 6, most of the function conforms quite nicely to the "short and vertical" model. If you're conditioned to look at algorithms in tiny bits, chances are you'll look at problems the same way-- which means you'll lose much of the power of APL.

Consider again, for a moment, the last two versions of the *Instructions for computing an average* given above, with attention to style. What situation can you conceive, in which you would express yourself that way? Version 3, in particular: you would never address another human being that way (well, save maybe your boss, if you *did* think that would get you a raise). You might address a machine that way. But, we hope, only when you didn't have any other choice. There are indeed many situations when you *do* have to address machines that way: when writing an APL interpreter, for instance. The tiny steps are in fact closer to how machines must execute our instructions, than to how human beings conceive of them. This is a partial answer to one of our questions: Who finds instructions easier when they're broken up into very short bits? Some machines do. APL doesn't require this because of its history: it was *not* originally defined to instruct machines. It is a *human* language.

But (tempting though it might be) we don't really feel it's fair to question the humanity of everyone who can't read the APL we write. Fortunately, it isn't necessary. There are human beings we might want to address with very short sentences, made out of a very limited vocabulary: people who don't speak the language we're using very well. People whose language *we* don't speak very well. Or, in writing: people of marginal literacy. People just learning to read.

Implied in these categories is the answer to our other question: in what way are instructions easier to understand when they're broken up into tiny bits? They're easier to understand in that you can focus on the meanings of the words themselves-- which you might want to do when you're not very sure of them.

In all the cases in which people find it easier to understand broken English, *it is generally expected that the problem is temporary*. For some reason, in the case of APL, we have trouble even admitting that this *is* the problem. In APL, when the literacy problem is recognized, it's usually attacked by limiting-- often voluntarily-- the style of those who *are* literate to the comprehension level of those who are not. The problem is, indeed, partly one of writing. Some people write English that's pretty hard to follow too. But at the level where it makes a critical difference to comprehension, to use only two or three words per sentence, *the problem is simply learning to read*.

This is also the case with APL. The major difference between APL and most other executable languages is simply that APL *has* a syntax sufficient to express thought; the others do not, and must use a sequence of steps instead. Any accompanying thoughts had better be expressed in comments, in these other languages, even if they're simply thoughts descriptive of the process. In APL, the best description of the thought is in the APL itself, and as in English, the thought can be expressed most concisely, directly, and meaningfully when we're not restricting ourselves to an illiterate audience.

Comments in English or some other natural language may still be desirable, but the useful ones are not descriptive, they're intentional: not *what* is this doing-- which the APL expresses better than English would-- but *why* is it doing it; or *to what* did the author expect to do it.

Let's look at a different real example, one published to argue for the precise opposite of the position we take in this paper. The accompanying figure lists the two functions from [3] mentioned at the outset.

Consider just the APL for the nonce, leaving aside the comments. We'll come back to those. *SPD* may look a little forbidding at first-- if you're not used to reading APL, or if you think you should read it in the order a computer will execute it. But there's no need for reading in that direction: in the culture APL was developed, human beings are used to reading from left to right-- and that's a fine way of reading APL. As Iverson remarks in a discussion of APL parsing rules,

> One important consequence of these rules is that in an unparenthesized sentence the right argument of any verb is the result of the entire phrase to the right of it. A sentence such as $3\times P\lceil Q\star\mid R-5$ can therefore be *read* from left to right; the overall result is three times the result of the remaining phrase, which is the maximum of P and the part following the $\lceil$, and so on. [4]

Let's read our example, then, from left to right, as we're accustomed to. The first thing that leaps to our attention is a parenthesis; we don't know much about what it encloses yet, but we notice that immediately to its right is a ⍴. We see immediately, then, that the result of *SPD* is a reshape of some value (giving a matrix result). Since what we're reshaping has just been transposed (reading on to the right), the parallel between the ⍉ arguments, and the left argument of the reshape, is very suggestive of collapsing two axes into one. This impression is reinforced by reading a little farther; the object transposed was itself the result of another reshape, and it in turn was the result of a ↑ on the right argument. A glance at the

```
    ∇ Z←COL SPD X;C;D;F
[1]  ⍝ SPREADS A RIGHT ARGUMENT MATRIX X OF SHAPE <M,N> TO A RESULT OF SHAPE
[2]  ⍝ <(T←⌈M÷COLS),COLS×N>. PLACES FIRST GROUP OF T ROWS IN THE FIRST N
[3]  ⍝ COLUMNS AND THE NEXT GROUP OF T ROWS IN NEXT N COLUMNS, ETC.
[4]  ⍝      FOR EXAMPLE: (3 SPD 3 4⍴'CAN YOU SEE ')↔1 12⍴'CAN YOU SEE '
[5]   Z←(F[2],×/F[1 3])⍴2 1 3⍉(F←COL,C÷COL,1)⍴(C←D+(COL|COL-(D←⍴X)[1]),0)↑X
    ∇

    ∇ Z←MULT SPREAD MAT;SHP;WIDTH;XSHP;MATPLUSROWS;NEW;F;M;TAR
[1]   ⍝ PROLOGUE:
[2]   ⍝            SPREADS A RIGHT ARGUMENT MATRIX TO A MULTIPLE
[3]   ⍝            OF ITS CURRENT WIDTH.  THE MULTIPLE IS SPECIFIED
[4]   ⍝            BY ITS LEFT ARGUMENT.
[5]   ⍝
[6]   ⍝      LEFT ARGUMENT: THE MULTIPLE DESIRED (AN INTEGER).
[7]   ⍝     RIGHT ARGUMENT: THE MATRIX TO BE SPREAD (USUALLY CHARACTER DATA).
[8]   ⍝
[9]   ⍝              RESULT: THE SPREAD MATRIX.
[10]  ⍝
[11]   SHP←⍴MAT ⍝                        THE SHAPE OF THE MATRIX
[12]   WIDTH←SHP[2] ⍝                    THE NUMBER OF COLUMNS IN THE MATRIX
[13]  ⍝
[14]   XSHP←SHP+(MULT|MULT-SHP[1]),0 ⍝XSHP IS THE SHAPE OF MAT WITH THE ROWS
[15]  ⍝                                  INCREASED TO A MULTIPLE OF MULT
[16]  ⍝
[17]   MATPLUSROWS←XSHP↑MAT ⍝            OVERTAKE MAT TO HAVE THIS NUMBER OF ROWS
[18]  ⍝
[19]   NEW←XSHP[1]÷MULT ⍝                NEW = THE NUMBER OF ROWS THE NEW MATRIX
[20]  ⍝                                  WILL HAVE. (THE MULTIPLE)
[21]  ⍝
[22]   F←MULT,NEW,WIDTH ⍝                SHAPE OF 3 DIM. INTERMEDIATE RESULT
[23]   M←F⍴MATPLUSROWS ⍝                 RESHAPE EXTENDED MATRIX TO 3 DIMENSIONS
[24]   TAR←2 1 3⍉M ⍝                     TRANSPOSE ARRAY SO MULT AND WIDTH ARE
[25]  ⍝                                  THE ROWS AND COLUMNS SO THAT RESULT IS
[26]   Z←(F[2],×/F[1 3])⍴TAR ⍝           ACHIEVED BY 'RAVELING' THE LAST 2 DIM.
    ∇
```

expression to the left of the ↑, though it may not reveal exactly what the value is, shows clearly that we're dealing with an overtake, since *D* is assigned right there as the shape of *X*, and (reading the parenthesized phrase also from left to right) we see that the ↑ argument is more than that. So: that's the whole function. We've just skimmed it, but this skimming tells us most of the story: the result is a matrix containing all of the argument *X*, but rearranged in some way, and also some padding.

For many purposes, we could stop right there: a little experimentation would tell us the rest we might need to know. But we can read more carefully, too, and discover as much detail as we need.

Look for a moment at the other version, *SPREAD*. Perhaps you can skim it as quickly as the first; we couldn't. There are rather more temporary variables involved to keep track of; the final reshape is not so suggestively associated with a transpose; the overtake is so buried and separated from *its* argument that, even knowing what it is and where it must be, we have trouble finding it.

But we didn't really finish reading the short *SPD*; in particular, the expression

```
C←D+(COL|COL-(D←⍴X)[1]),0
```

seemed rather mysterious. Notice, however, line [14] of the long *SPREAD*-- it has exactly the same expression! Slightly different variable names, and the ⍴*X* assignment has been moved, making it a little harder to see what one of the variables is... but no substantial difference. The rewrite, in other words, did absolutely nothing to clarify the one obscure part of this function. It is fairly clear that breaking this up further wouldn't do it. What does this do? Well, there are two approaches-- one could try to simplify and analyze with no idea of where one was going; or one could take some knowledge of the intention of the function, together with reading of the rest of the function, to form a conjecture, which could then be verified. We took the second path; here's where comments come in handy-- we needed to know the author's intent. Knowing the purpose of both functions (the comments on the first seem a little more helpful here), it was easy to conjecture that the overtake must be to pad the original data *X* to a number of rows which is an even multiple of the number of "logical columns" desired, *COL*-- therefore, our conjecture was that this phrase must be equivalent to

```
C←(COL×⌈(1↑D)÷COL),1↓D←⍴X
```

A short proof verified that this was indeed the case. Note that the "skimming" we went through in the first place was crucial to form the conjecture (which made the analysis, we suspect, much shorter than if we'd had no idea where we were going). This skimming, as we began by showing, is much easier when all the context is immediately in front of us. That is, for the one part of both functions that is hard to understand, *SPD*-- the "nasty" one-liner-- makes it easier to discover the meaning than the broken-up *SPREAD*.

Our central conclusion was arrived at before examining this example: that the problem of written communications, in APL as in English, requires skills on the part of the reader as well as on the part of the writer. The responsibility for

communication has been laid too heavily on writers in the APL community.

Modifying one's APL writing style to cater to an illiterate audience has been a recommended approach. We have tried to show, first, that becoming accustomed to a less expressive style of writing can hamper a writer's thinking in approaching a problem; second, that reading skills are valuable in themselves; and, finally, that -- assuming readers of APL are willing and able to read -- the often-recommended "short and vertical" style makes it harder, rather than easier, to read APL, especially when obscure phrases are encountered in either style.

As with any aesthetic issue, the question of good style in APL cannot be settled prescriptively. A final, and perhaps the weightiest, reason for developing APL reading skills is that to read is, in the final analysis, the most sensible advice one can give to writers concerned with improving their own style. Iverson has remarked that

> Perhaps the most important habit in the development of good style in a language remains to be mentioned, the habit of critical reading. Such reading should not be limited to collections of well-turned and useful phrases. . .nor should it be limited to topics in a reader's particular specialty.
>
> . . .one may benefit from the critical reading of mediocre writing as well as good; good writing may present new turns of phrase, but mediocre writing may spur the reader to improve upon it. [5]

Perlis and Rugaber[6] have advocated teaching the recognitition of particular phrases (often called "idioms" in the APL community) as a useful step in teaching both reading, and writing, of APL. Published collections of such phrases include Perlis and Rugaber's report *The APL Idiom List*[7] and the more recent *FinnAPL Idiom Library*[8]. As Iverson remarks, such collections are certainly one kind of useful reading matter. But by themselves they are unlikely to make anyone literate, and in fact careless use of such collections in teaching can sometimes disguise illiteracy rather than promote literacy, if students feel encouraged to simply accept, recognize, and copy such phrases rather than actually reading and analyzing them (see Pesch[9] for more discussion of this issue).

We can provide no easy answers: but this much is clear-- writers of APL must assume a literate audience (as writers of English do) if they are to use the language effectively; and readers of APL (which is to say all of us) can read best by reading more. In the end, greater literacy is its own reward.

References.

[1] Jonathan Amsterdam, "Computer Languages of the Future", *Popular Computing*, September 1983

[2] "APL Quote Quad Style Guide for APL Functions", *APL Quote Quad* 15 3, March 1985, p. 8

[3] Eugene R. Mannacio, *Standards for APL Applications Development and Enhancement* (Fireman's Fund American Life Insurance Co., July 2, 1984)

[4] K. E. Iverson, *A Dictionary of the APL Language* (I.P. Sharp Associates, DRAFT, 5 September 1985)

[5] K. E. Iverson, "Programming Style in APL", *An APL Users Meeting* (1978), Proceedings (I.P. Sharp Associates, September 1978)

[6] Alan J. Perlis and Spencer Rugaber, "Programming with Idioms in APL", *APL79 Conference Proceedings* (*APL Quote Quad* 9 4, June 1979)

[7] Alan J. Perlis and Spencer Rugaber, *The APL Idiom List* (Yale University Technical Report #87, April 1977)

[8] *FinnAPL Idiom Library* (Finnish APL Association; second edition, 1982)

[9] Roland H. Pesch, Review of *FinnAPL Idiom Library* (*APL Quote Quad* 13 2, December 1982)